

Regular Expressions with .NET

By Dan Appleman

1st Edition – February 2002

Revised April 2003

Copyright © Daniel Appleman 2002-2003

All rights reserved

Published by Daniel Appleman in cooperation with Desaware Inc.

www.desaware.com

Regular Expressions with .NET

With the release of Visual Studio .NET, a great deal of attention has been placed on the Visual Studio .NET languages, Visual Basic .NET, C# and Managed C++ (not to mention the dozens of others under development by various companies). It might surprise you to know that yet another language is built into Visual Studio – one that can be used in conjunction with VB .NET or any other .NET language. A language that is terse to such a degree that the term “concise” does not come close to describing its brevity of syntax. A language so cryptic that it can take hours to truly understand a single line of code.

Yet it is a language that can save you hours upon hours of time in any application that involves text processing. It is a language that can perform complex data validation tasks in a single line of code. It is a language that performs sophisticated search and replace operations on strings. It is a language that should be part of every programmer’s “bag of tricks.”

I am talking about “Regular Expressions” – a language designed to parse and manipulate blocks of text.

This ebook is intended to be a complete introduction to Regular Expressions that can even be read and understood by programmers who have never heard of them. It is also intended to help experienced Regular Expression programmers come up to speed quickly on the .NET implementation of Regular Expressions.

Author’s Bio:

Daniel Appleman is the president of [Desaware Inc.](#), a developer of add-on products and components for Microsoft Visual Studio, including [CAS/Tester](#), [SpyWorks](#), [StateCoder](#) and the [NT Service Toolkit](#) for .NET languages and VB6. He is a cofounder of [APress](#), a publishing company specializing in high quality professional level books for computer programmers and Information Technology professionals. He is the author of numerous books including "[Moving to VB.NET: Strategies, Concepts and Code](#)", "[How Computer Programming Works](#)" and "[Dan Appleman's Visual Basic Programmer's Guide to the Win32 API](#)" and he is the author of a new series of [Ebooks](#) on .NET related topics.

Stop! – Before You Read Further

This E-Book is sold on a per-reader basis for only \$14.95.

If you have already purchased this book online or from Desaware, thank you.

However, if you have obtained the book through other channels, I would appreciate it if you would pay for it using the Amazon honor system. Your support makes it possible for me to continue to write E-Books. I feel that an E-Book, or E-Doc (as Amazon calls them) in the 25-100 page range is the perfect length for many subjects – too long for a magazine, but too short for a book.



What should you pay?

- The recommended price is \$14.95
- If you really can't afford it (in high school, or unemployed), pay less, or pay when you can.
- If you are not satisfied, pay nothing.

What can you do with it?

The \$14.95 gives you the right to read this book unlimited times, make backups, and install it on as many of your machines as you wish for your personal use.

Think of it has a hybrid between a book and shareware. And thank you for your support.

Table of Contents

Introduction	1
Sample Code	2
Part I – The Basics	3
Introduction to Regular Expressions	3
Learning Regular Expressions	6
Regular Expression Patterns	6
First rule for Regular Expression Patterns:	7
Second rule for Regular Expression Patterns:	7
Escapes and special characters:	7
Quantifiers and Alternates	8
Character sets	10
Grouping and Backreferences	10
Regular Expression Operations	12
Finding Matches	12
Search and Replace	14
Splitting a String	16
Data Validation	18
Part II - Regular Expression Objects in .NET	19
The Regex class	19
Creating and using a Regex class:	19
Using Static Regex methods	20
Regex Class Options	21
Ignore Case Option	21
SingleLine and MultiLine Options	21
ExplicitCapture Option	23
IgnorePatternWhitespace Option	23
RightToLeft Option	25
ECMAScript Option	25
Compiled Option	26
Groups and Captures	26
The RegexTester example	27
Groups in Depth	31
Captures in Depth	34
Part III - Advanced Regular Expressions	38
Zero-Width Assertions	38
\A, \Z and \z	38
\b and \B	39

\G	39
Zero Width Pattern Assertions	41
More on Quantifiers	43
More on Grouping	44
Balancing Group Definitions	44
Non-Backtracking Constructs	50
Advanced Search and Replace	52
Part IV - Additional Topics	54
Compiling Regular Expressions	54
Performance Considerations	55
Threading Issues	56
Part IV - What are State Machines, and Why Should You Care?	57
Why are State Machines Important?	58
Part V - Conclusion	62
Index	63
Appendix A – Regular Expression Pattern Reference	67
Single Character Escapes	67
Assertions	68
Grouping and Backreferences	68
Quantifiers and Alternating constructs	69
Replacement Text	69
Comments	69
Appendix B - Books and Products by Dan Appleman	70
Software	70
Books by Dan Appleman	72
eBooks by Dan Appleman	72
Appendix C - Publishing	75

Introduction

This ebook is intended to be a complete introduction to Regular Expressions that can even be read and understood by programmers who have never heard of them. It is also intended to help experienced Regular Expression programmers come up to speed quickly on the .NET implementation of Regular Expressions.

My focus will be to help you gain a strong enough understanding of Regular Expressions to be able to use relatively simple expressions frequently in your day to day programming efforts. For example: while advanced computer scientists might be especially interested in creating complex Regular Expressions for language parsing tasks, I'm more interested in helping beginning and intermediate .NET programmers use them in their daily routines for tasks such as input string data validation or smart data substitution in strings. Despite my "beginner/intermediate" focus, this ebook covers all of the .NET regular expression constructs.

All code samples are provided in both Visual Basic .NET and C#.

In order to provide the necessary depth, while not scaring off beginners, this ebook is divided into three main parts.

- Part I An introduction to Regular Expressions, and coverage of the most commonly used escapes and pattern constructs.
- Part II Covers the .NET Framework Regular Expression object model, demonstrating the use of all .NET Regular Expression objects.
- Part III Covers more advanced Regular Expression concepts and constructs. Beginners and intermediate programmers may find that they will never use the material covered in this part of the book.
- Part IV Provides some insight into state machines, the methodology on which Regular Expression engines are based.

While this ebook does cover the material in the MSDN documentation, I think you will find it anything but a manual rehash. The Microsoft documentation is extremely terse, and in some cases nearly incomprehensible – especially to those new to Regular Expressions. Aside from including samples that illustrate every non-trivial construct, I've hopefully improved on their explanations as well, and have followed a tutorial style in which each section builds on the next (instead of throwing everything at you all at once).

As mentioned earlier, this ebook is licensed per user – treat it like a software product. I chose the ebook format because, at 60+ pages, it is far too long to publish as a magazine article. Yet it is too short to publish in a printed book (though you are welcome to print out a copy for your own use). I believe that ebooks are ideal for works in the 25-100 page length, and by paying for your copy, you ensure that more such works will become available both from me and from other authors.

And now, let us begin.

Dan Appleman

dan@desaware.com

February 2002.

Sample Code

You can download the sample code for this ebook from

<ftp://ftp.desaware.com/ebooks/regexebook.zip>

When you unzip the file, be sure to specify that the directory structure should be preserved.

The sample programs are provided in both Visual Basic .NET and C# versions, and are compatible with the final release of Visual Studio .NET.

Important Note!

You are strongly encouraged to download the sample code from our FTP site rather than trying to type in the code into your own projects. Aside from the possibility of errors occurring as you type in the code, the samples here do not include all of the details required to run the code, such as project settings and namespace imports.

Part I – The Basics

Consider this example:

Let's say you have a string that contains a page of HTML text you retrieved from a web site. Say you want to extract all of the headers on the page. You could write VB code to do this, but think of what you need to do:

- You'll need to write code to identify HTML tags (HTML terms contained between <> brackets).
- You have to check each tag you find for the letter H followed by a digit, followed either by a closing bracket or additional formatting information.
- You need to find the equivalent closing tag – consisting of the </Hn> tag, where n is the same digit as you found earlier.
- You then need to extract the text between the two tags.

True, this is not a huge amount of work. But you could easily spend an hour or two writing and testing the necessary code.

Or you could do all of this using the following line of Regular Expression code:

[VB]

```
Results = Regex.Matches(inputtext, _
    "<(?!<hdr>(h|H)\d).*?>(?!<result>.*)</\k<hdr>>")
```

[C#]

```
Results = Regex.Matches(inputtext, _
    @"<(?!<hdr>(h|H)\d).*?>(?!<result>.*)</\k<hdr>>");
```

At which point, if you are unfamiliar with Regular Expressions, your mouth drops open and you stare in shock at what is undoubtedly one of the most cryptic lines of code you have ever seen.

We'll come back to this line of code later. Let's start at the beginning.

Introduction to Regular Expressions

A Regular Expression processor is an interpreter that uses a pattern to parse a string of text, or a compiler that produces code that is able to parse a string of text. The .NET Regular Expression processor works both ways, generally acting as an interpreter but also able to compile an assembly for expressions that are to be reused frequently.

Parsing text consists of breaking text up into components. For example: if you wanted to convert a sentence into words, you would look for the spaces that separate the words.

Consider the sentence "This is a sentence".

Each word consists of one or more characters, each followed by one or more spaces or the end of the line (we'll ignore punctuation for now).

In VB, you might use a loop and the Instr function to look for spaces and extract the words. But you can also use the following Regular Expression for a word (again, we're ignoring punctuation):

```
\w+(\s+|$)
```

This expression breaks down as follows:

\w	A character or digit (including the underscore character)
+	One or more of whatever precedes it (in this case characters or digits)
(	A group consisting of...
\s	A white space character
+	One or more of whatever precedes it (in this case a white space character)
	or
\$	The end of the string
)	The end of the group

In English: One or more letters, followed by one or more spaces or the end of the line.

The RegexIntro example program is a simple example for demonstrating the use of regular expressions (you'll see a more sophisticated example later). Enter the sentence to parse in the upper text box. Then select the Parse-Words menu command.

This calls the function ParseText as follows:

[VB]

```
Private Sub mnuParse_Words_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles mnuParse_Words.Click  
    ParseText("\w+(\s+|$)", Nothing)  
End Sub
```

[C#]

```
private void mnuParse_Words_Click(object sender, System.EventArgs e)  
{  
    ParseText(@"\w+(\s+|$)", null);  
}
```

The ParseText function is defined as follows:

[VB]

```
' Parse text using the specified regular expressions. Display any
' groups with the name GroupToShow
Private Sub ParseText(ByVal pattern As String, _
    ByVal GroupToShow As String)
    Dim mc As MatchCollection
    mc = Regex.Matches(TextBox1.Text, pattern)
    Dim m As Match
    ListBox1.Items.Clear()
    For Each m In mc
        ListBox1.Items.Add(m.Value)
        If GroupToShow <> "" Then
            If m.Groups.Item(GroupToShow).Value <> "" Then
                ListBox1.Items.Add("result: " & _
                    m.Groups.Item(GroupToShow).Value)
            End If
        End If
    Next
End Sub
```

[C#]

```
// Parse text using the specified regular expressions.
// Display any groups with the name GroupToShow
private void ParseText(string pattern, string GroupToShow)
{
    MatchCollection mc;
    mc = Regex.Matches(textBox1.Text, pattern);
    listBox1.Items.Clear();
    foreach (Match m in mc)
    {
        listBox1.Items.Add(m.Value);
        if (GroupToShow!=null)
        {
            if (m.Groups[GroupToShow].Value!=null)
            {
                listBox1.Items.Add("result: " +
                    m.Groups[GroupToShow].Value);
            }
        }
    }
}
```

The GroupToShow parameter contains the name of a group to display. I'll talk about that shortly. The Regex.Matches method is a static method of the Regex class, which in turn is defined in the System.Text.RegularExpressions namespace. The method returns a collection of Match objects. When the Matches method is called, the Regular Expression processor scans through the string looking for text that matches the pattern specified. In

this case it looks for sequences of letters followed by white space. Each time it finds a match, the method creates a Match object and adds it to the Matches collection.

Don't worry if this is a bit confusing, I'll be covering the objects of the System.Text.RegularExpressions namespace in more depth later in this document.

The ParseText routine then iterates through the mc collection, displaying each match.

If there is a group in the match that has the name specified by the GroupToShow parameter, that group is displayed as well. I'll discuss that in more detail shortly.

Learning Regular Expressions

As with any computer language, it will take you some time and study to become really proficient with Regular Expressions. My goal here is not to provide you with a complete reference to .NET Regular expressions – you'll find that in Appendix A and, of course in the online documentation. Instead, my goals are:

- To introduce the idea of Regular Expressions to readers who may be completely new to the concept, and to demonstrate the practical uses of this technology.
- To translate Microsoft's occasionally incomprehensible documentation into something resembling English.
- To provide a clear and concise explanation of the .NET Framework implementation of Regular Expressions, including the key classes and methods you will be using.

One thing to keep in mind - even if you are already familiar with Regular Expressions from other applications, is that the Regular Expression language is far from standard¹. It varies from implementation to implementation. This document will only cover Regular Expressions as implemented by the .NET Framework. Fortunately, you'll find that it is an exceptionally powerful implementation, and most of what you learn will be directly applicable to other platforms.

Regular Expression Patterns

One term that you will see often when reading about Regular Expressions is the term "match". The idea here is that a Regular Expression engine is searching through an input string, searching for text that matches the specified condition.

Consider the Regular Expression pattern:

`A\w+`

This expression breaks down as follows:

- | | |
|-----------------|---|
| <code>A</code> | The letter A |
| <code>\w</code> | A character or digit (including the underscore character) |
| <code>+</code> | One or more of whatever precedes it (in this case characters or digits) |

In English: Any word beginning with the letter A.

¹ Actually, the issue of standards is somewhat complex. I'll discuss this later in section titled Regex Class Options.

In other words, when the Regular Expression engine scans the input string, it will detect any place in the string where it finds a word beginning with the letter A.

Try entering the following input string:

```
Apple Banana Orange Apricot
```

Then, using the Parse-User menu command, enter the pattern `A\w+`.

The result will show:

```
Apple
Apricot
```

The Regular Expression engine determined that the substrings “Apple” and “Apricot” in the input string *matched* the Regular Expression pattern.

You’ll see the term “match” in the context of Regular Expressions to describe an element of a pattern matching an element in the input string.

Thus, in the case of `A\w+`, you would say that:

- ‘A’ matches all occurrences of the letter A
- `\w` matches all characters or digits.
- `A\w+` matches all words beginning with the letter A.

Most of working with Regular Expressions consists of coming up with patterns that perform the matching operation that you are looking for. That’s the subject for the rest of this section.

First rule for Regular Expression Patterns:

If the character is not one of `. $ ^ { [ ( | ) * + ? \` it is simply an element in the pattern.

For example: the pattern “Hello” will match every appearance of the word “Hello” in a string. Try using the RegexIntro example with the sentence “Hello, anyone there?, Hello?” and the pattern “Hello”. You will see two matches.

Second rule for Regular Expression Patterns:

If you want to use any of the special characters in the first rule as part of a pattern, precede them with the `\` character. Thus `\$` matches a dollar sign.

Escapes and special characters:

The `\` character is a prefix that gives some characters special meanings. For example:

```
\n matches a newline (LF) character
\r matches a return (CR) character
\t matches a tab character
\w matches a character (a-z, A-Z, 0-9 and underscore)
\W matches any character that is not a letter.
\s matches any white spaces (space or tab)
\S matches any character that is not white space
\d matches a digit (0-9)
```

`\D` matches any character that is not a digit
`.` matches any character other than the end of line or end of text
`^` matches the beginning of a string or line
`$` matches the end of the string or line
`\b` matches the boundary of a word
`\B` matches anything that is not the boundary of a word

A complete list of character escapes can be found in Appendix A.

Important Note!!

When you are using C#, it is important that you remember that the Regular Expression patterns must contain the `\` character itself to provide the escape. The C# compiler itself uses the `\` character as an escape. So, you were to use the literal string `"\n"` in a C# expression, the result will be to pass a single newline character as the pattern, and not the two character string consisting of the backslash followed by n. In C# you can use either of the two literal strings to obtain a `"\n"` pattern:

```
"\\n"    // in which \\ is converted into a single backslash  
@"\n"    // in which the @ symbol disables the escaping  
mechanism.
```

Throughout this ebook, code examples are provided for both languages. However, all pattern strings in the text (outside of code listings) will be the actual Regular Expression pattern, not including the C# escape syntax.

Quantifiers and Alternates

You can append the following special characters to indicate a repetition of the previous character or group. For example:

- * Repeat zero or more times matching as many characters as possible.
- + Repeat one or more times matching as many characters as possible.
- ? Repeat zero or one time matching as many characters as possible
- *? Repeat zero or more times matching as few characters as possible.
- +? Repeat one or more times matching as few characters as possible.
- | When between two characters or groups, matches one or the other (this is called an alternating operation, because it chooses among two alternatives).

A complete list of quantifiers and alternating operators can be found in Appendix A

The idea of matching as few as possible or as many as possible can be a bit confusing at first. Here's an example that might help clarify what's going on.

Let's say you want to identify sentences in a block of text. A sentence is defined as any series of characters followed by a period and a space, or followed by a period and the end of the text.

The following pattern can be used:

```
.*\.( |$)
```

This expression breaks down as follows:

- . Any character
- * Zero or more of whatever precedes it (in this case any character)
- \. A period
- (Start of a group consisting of
- sp | \$ A space or the end of the text (sp indicates the space character in this text and is not a Regular Expression term).
-) End of the group

In English: Zero or more characters, followed by a period and space, or period (at the end of the text).

Using the RegexIntro program, enter the following line as the input text:

```
This is a sentence. This is another sentence.
```

When you test against the pattern `.*\.( |$)` using the `Parse_User` menu command, the result will be:

```
This is a sentence. This is another sentence.
```

Why didn't it find the first sentence?

The problem is that the period at the end of the first sentence can match two ways – either as a period at the end of a sentence, or as a character within a sentence (i.e. the period matches both `.` and `\.` in the pattern).

When the Regular Expression engine tries to match the pattern, it sees two possible ways to match the sentence. It can match the `\.` at the first period, or at the second. In this case it chooses the larger match.

To specify that you want the smallest possible match, change the pattern to:

```
.*?\.( |$)
```

The `*?` quantifier changes the match to request the smallest possible number of characters that match. The result will be to match the text up to first period. The remaining text will then be a second match. The result will be the following two matches:

```
This is a sentence.  
This is another sentence.
```

Additional quantifiers can be found in Appendix A that allow you to specify a minimum, maximum or range of characters required for a valid match.

Character sets

You can also define sets of characters by placing them in square brackets.

For example:

[aeiouAEIOU] matches all upper and lower case vowels.

[a-z] matches all lower case letters.

[^abc] matches every character except for a, b and c.

For example:

Let's say you want to find every word in the text that begins with a capital letter.

You earlier used the pattern `"\w+(\s+|$)"` to find all of the words in a string.

Now, using the RegexIntro example, enter the text

```
This is another Line of text
```

And use the pattern `"[A-Z]\w*(\s+|$)"`

In the list box you'll see the results:

```
This  
Line
```

To find any word we used the term `\w+` which returns one or more letters. This has been replaced with `[A-Z]\w*` which breaks down as follows:

[A-Z] Any one letter from A-Z (capitalized).

\w Any letter (upper or lower case)

* Zero or more of the preceding (zero or more letters).

It is important to change the term after the `\w` character from `+` (one or more) to `*` (zero or more) so that the pattern will correctly match single character words such as A and I.

Grouping and Backreferences

You can group patterns by placing them in parenthesis. You can give a name to the group as well. Groups serve a number of purposes.

- They can make a Regular Expression much easier to read.
- Quantifiers that follow a group apply to the entire group
- Groups within a match can be identified by group number or by name – allowing you to extract information from within a matched string.

That allows you to isolate the part of the string that matched the group from the entire match. Here are some of the grouping constructs you'll be using.

- () Defines a simple group.
- (?<name>) Group named "name"
- (?i:) Ignore case when matching within the group
- \n Matches a previous group (group # n)
- \k<name> Matches a previous group with the specified name.

Groups that don't have a name, have a number. Consider the pattern used to find words earlier:

```
\w+(\s+|$)
```

Let's modify it slightly as follows:

```
(\w+)(\s+|$)
```

Now you have two unnamed groups. The first is group 1, the second group 2 (group zero always corresponds to the entire match, and unnamed groups are otherwise numbered left to right in order of their opening parenthesis). When the match takes place, the portion of the input text that was assigned to the group is said to have been "captured" into the group. When examining the results of the match, you can extract the captured values from individual groups. In this case this would let you examine the word without the white space characters. The word is captured by the `(\w+)` term, and the white space between words is captured by the `(\s+|$)` term.

Backreferencing allows you to match a previous group. For example, the pattern:

```
\b(\w+)(\s+|$)\1
```

This expression breaks down as follows:

- \b Matches the start of a word
- (\w+) A group consisting of one or more characters (letters, digits or underscore). This will be group #1
- (\s+|\$) A group consisting of one or more white space characters, or the end of the line). This will be group #2
- \1 Matches whatever was found in group #1

Additional Grouping options can be found in the Advanced Regular Expressions section later in this Ebook, and in Appendix A.

Using the RegexIntro program, try using this pattern with the sentence:

```
This is a a sentence.
```

The result will be

```
a a
```

Why?

The Regular Expression engine starts at each word boundary. The `(\w+)(\s+|$)` pattern will match every word. The `\1` pattern will only match whatever was found in group 1. In other words, only repeated words will be found.

Why did we have to use \b to set the initial word boundary?

Try the same sentence without the \b.

The result is:

```
is is
a a
```

But wait, “is” isn’t a repeated word, is it?

It isn’t – but it does match. You’re seeing the match “**This is** a a sentence.”

Because you didn’t specify that the match must start at a word boundary, the pattern matched any place where the ends of words match.

You may be wondering by now how it’s possible for anyone to figure out the right patterns to perform a particular task.

Practice helps, but I assure you that even programmers with a great deal of experience using Regular Expressions spend a lot of time experimenting. Trial and Error is a useful tool indeed when it comes to figuring out the patterns you need.

Regular Expression Operations

At this point, let’s take a few minutes to illustrate how what you’ve learned so far can be used to perform rather complex tasks.

Finding Matches

Microsoft includes a powerful XML parser with .NET, which is fine if you’re parsing well-formed XML. But parsing HTML can be a bit trickier. Sure, you can use Internet Explorer and its document object model to explore an HTML page. But for quickly extracting information from an HTML page, regular expressions provide a fast and powerful solution.

Before looking at the rather complex header example, let’s look at a regular expression that can extract the title of an HTML page. For testing these patterns, browse to the page of your choice using any browser, then use the browser’s “View Source” command to retrieve the raw HTML for the page and copy it to the clipboard. You can then paste the HTML into the RegexIntro sample project’s text box.

The Parse-Title menu command uses the following pattern to find the titles in the HTML text:

```
<(?:Title)>(?:<result>.*)</(?:title)>
```

Let’s break this down:

<	Opening tag bracket
(?:	Start group, ignore case
Title	The word “Title”
)	Close group
>	Closing tag bracket
(?:<result>	Open a group named “result”

.*	Zero or more characters
)	Close the “result” group
<	Opening tag bracket
/	/ character (not to be confused with the \ escape character)
(?i:	Start group, ignore case
title	The word “title” (note, we don’t care about case)
)	Close group
>	Closing tag bracket.

In plain English: Look for an opening <title> tag followed by some text and a closing </title> tag. Ignore the case of the word “title”. Take all the text between the tags, and place it in a group named “result”.

In the ParseText function, you may recall the text:

[VB]

```
If GroupToShow <> "" Then
    If m.Groups.Item(GroupToShow).Value <> "" Then
        ListBox1.Items.Add("result: " &
            m.Groups.Item(GroupToShow).Value)
    End If
End If
```

[C#]

```
if (GroupToShow!=null)
{
    if (m.Groups[GroupToShow].Value!=null)
    {
        listBox1.Items.Add("result: " +
            m.Groups[GroupToShow].Value);
    }
}
```

The GroupToShow parameter in this case contains the string “result”. If a group named “result” is found in the Groups collection, it is displayed as well.

Why use a named group? The pattern as a whole matches the <title> and </title> tags and the information between the tags – both the tags and the information between them is part of the match. The named group provides us an easy mechanism to extract the data between the tags – which is what you’re probably interested in.

Now let's take a look at the header pattern that you saw at the start of this article.

```
<(?(<hdr>(h|H)\d).*?)(?(<result>.*</\k<hdr>>
```

Let's break this down:

<	Opening tag bracket
(?(<hdr>	Start of a group named "header"
(h H)	Group consisting of upper or lower case H
\d	Any digit
)	Close the "hdr" group (which contains Hn, where n is the header number)
.*?	Zero or more of any character, matching as few as possible until the next...
>	Closing tag bracket.
(?(<result>	Start a group named "result"
.*	Zero or more characters
)	Close the "result" group.
<	Opening tag bracket
/	The / character
\k<hdr>	Matches the group named hdr. If H3 was found earlier, this will match H3
>	The closing tag bracket.

In English:

Search for a string that starts with a header tag consisting of <Hn> or <Hn formatting info>, followed by arbitrary information, followed by a closing tag </Hn> where n of the opening tag matches that of the closing tag.

The trick with the \k option is called "backreferencing", where the Regular Expression processor can create a match based on group information generated on the fly. A backreference matches the specified group.

This pattern will result in matches for all text within headers on the HTML page.

Search and Replace

As you've seen, Regular Expressions are most commonly used to parse string data – dividing a string into components based on a Regular Expression pattern. But it turns out that there is another equally useful purpose for Regular Expressions. They are phenomenal tools for perform search and replace operations in strings.

The RegexIntro example includes the Parse-Replace menu command that allows you to experiment with search and replace operations. The code for this command is simple:

[VB]

```
Private Sub mnuReplace_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles mnuReplace.Click
    Dim replaceForm As New frmReplace()
    replaceForm.ShowDialog()
    MessageBox.Show(Regex.Replace(TextBox1.Text, replaceForm.Pattern, _
        replaceForm.ReplaceString).ToString, "Result", _
        MessageBoxButtons.OK)
    replaceForm.Dispose()
End Sub
```

[C#]

```
private void mnuReplace_Click(object sender, System.EventArgs e)
{
    frmReplace replaceForm = new frmReplace();
    replaceForm.ShowDialog();
    MessageBox.Show(Regex.Replace(textBox1.Text, replaceForm.Pattern,
        replaceForm.ReplaceString).ToString() , "Result",
        MessageBoxButtons.OK );
    replaceForm.Dispose();
}
```

The `frmReplace` class contains two text boxes, whose contents can be read using the form's `Pattern` and `ReplaceString` properties. The `Replace` method used here is a static method of the `Regex` class – you'll read more about this later.

Let's start by looking at a simple example.

Enter the following text string into the input textbox:

```
This is a string
```

Then, using the `Parse-Replace` command, use the pattern `\s`, and the replace string `_` (underscore character).

The resulting text is:

```
This_is_a_string.
```

At first glance, while more powerful than the `System.String.Replace` method (because of the more sophisticated pattern matching), this may not seem all that useful. But try the following pattern:

```
(\s*)Dim\s+(\w+)\s+As\s+(\w+)
```

combined with the following `Replace` string:

```
$1$3 $2;
```

When applied to the following input string:

```
Dim xyz As Integer
```

The result is:

```
Integer xyz;
```

Wow – a one line program that converts simple Visual Basic .NET style variable declarations into the equivalent C# variable declaration. And yes, you can extend the pattern to handle more complex conversions, such as those that include initialization text.

Let's take a closer look at the pattern:

```
(\s+)Dim\s+(\w+)\s+As\s+(\w+)
```

This expression breaks down as follows:

- (\s+) This group matches all the leading spaces before the Dim statement and captures them into group #1.
- Dim Matches the word "Dim"
- \s+ Matches any number of spaces.
- (\w+) Matches the variable name and captures it into group #2.
- \s+ Matches any number of spaces
- As Matches the word "As"
- \s+ Matches any number of spaces.
- (\w+) Matches the variable type and captures it into group #3.

Now look at the replacement string:

```
$1$3 $2;
```

In replacement strings, a \$ is a special character indicating that you wish to include a captured group in the replacement string. This can take the form \$n, where n is the group number, or \${name} where name is a named group.

In this case, the replace string breaks down as follows:

- \$1 Insert group #1 (the leading spaces for the line)
- \$3 Insert group #3 (the type of variable)
- sp Insert a space (sp used to indicate a space in this text only)
- \$2 Insert group #2 (the variable name)
- ; Add a semicolon at the end of the line

The Regular Expression search and replace capability is a powerful feature for not only finding patterns, but creating "smart" replacement patterns that build on and rearrange information from the source text. In fact, it is even possible to specify a delegate to be called with each match, allowing you to programmatically determine the substitution. You'll see an example of this in the Advanced Regular Expressions section of this ebook.

Splitting a String

Regular Expressions can also be used to divide a string into substrings. This operation is similar to the System.String.Split method, but uses Regular Expressions to determine the separator pattern.

The RegexIntro example includes the Parse-Split menu command that allows you to experiment with split operations. The code for this command is simple:

[VB]

```
Private Sub mnuSplit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles mnuSplit.Click
    Dim s As String
    Dim ResultArray() As String
    s = InputBox("Enter Regex pattern for Split")
    ResultArray = Regex.Split(TextBox1.Text, s)
    ListBox1.Items.Clear()
    ListBox1.Items.AddRange(ResultArray)
End Sub
```

[C#]

```
private void mnuSplit_Click(object sender, System.EventArgs e)
{
    string s;
    string[] ResultArray;
    frmInputBox ibox = new frmInputBox();
    ibox.ShowDialog();
    s = ibox.textBox1.Text;
    ibox.Dispose();
    ResultArray = Regex.Split(textBox1.Text, s);
    listBox1.Items.Clear();
    listBox1.Items.AddRange(ResultArray);
}
```

It's not unusual to see commas used as delimiters in tables. The CSV (comma separated value) format takes this approach.

Try entering the following line in the text input box:

```
First field,Second field,Third field
```

Note the lack of a space between the first and second fields, and the extra spaces between the second and third fields.

Select the Parse-Split menu command, and enter the following pattern string:

```
,\s*
```

This expression breaks down as follows:

```
,      The comma character
\s*    Zero or more white space characters
```

The result will be the following array of three strings:

```
First field
Second field
Third field
```

The pattern string in this case captures not only the comma, but any white space that follows. The matched strings are considered delimiters, and the text between them is returned in an array. The Regular Expression Split command allows you to split strings with more flexibility than the System.String.Split method, because you can build patterns that accept (or tolerate) variation in the delimiter pattern. You'll see a more advanced CSV type example in the Advanced Regular Expressions section of this ebook.

Data Validation

One of the most useful features of Regular Expressions is in data validation. Validation that would otherwise take complex coding and testing can be replaced with a single line of code.

Consider this pattern:

```
^((\(\d\d\d\))|(\d\d\d))[- ]\d\d\d-?\d\d\d\d
```

This expression breaks down as follows:

<code>^</code>	Matches the start of the input string
<code>(</code>	Opens a group
<code>(\(\d\d\d\))</code>	Matches three digits in parenthesis. Note the use of the escape <code>\(</code> and <code>\)</code> to specify that you wish to use match the paren characters rather than open a group.
<code> </code>	Or match
<code>(\d\d\d)</code>	Match any three digits
<code>)</code>	Closes the group consisting of three digits (in parenthesis or not)
<code>[-]</code>	Matches a space or – character.
<code>\d\d\d</code>	Matches any three digits
<code>-?</code>	Matches zero or one – characters (i.e., the dash is optional)
<code>\d\d\d\d</code>	Matches any four digits

This pattern will match several formats of U.S. phone numbers including the area code. Try it using the Regex Intro program using the Parse_User menu command.

As an exercise, try improving the pattern to do the following:

- Make the area code optional.
- Make sure the first digit of the area code and the first digit of the 3 digit number prefix is not zero.
- Add an option to capture an extension in the format “x ...” or “ext ...”
- Try writing code to perform phone number validation and see how much longer it takes!

Part II - Regular Expression Objects in .NET

So far you've learned the basics of the Regular Expression language. You haven't seen all of the escapes and constructs available (those can be found in Appendix A and in the "Advanced Regular Expressions" section later in this Ebook). But you've seen the constructs that you'll be using most often, and you've seen some examples of the power of Regular Expressions for processing text.

Most important, you know enough now for us to take a closer look at how to use the .NET Framework classes that implement Regular Expressions in .NET.

All of the classes that implement Regular Expressions in .NET can be found in the `System.Text.RegularExpressions` namespace.

The Regex class

The `Regex` class is the main class for working with Regular Expressions in .NET. In its simplest form, the `Regex` class takes a pattern and some input data, then determines the Regular Expression match or matches that result. As you saw earlier, the `Regex` class is also able to perform Regular Expression based search and replace operations and can divide a string into substrings based on a Regular Expression pattern.

There are two approaches for using the `Regex` class.

- You can create an instance of the class, then call a method to perform the desired operation.
- You can call a static method on the `Regex` class to perform the desired operation

Here are some examples of how you can divide a string into words using the word break pattern (with which you are already familiar):

```
\w+(\s+|$)
```

Creating and using a Regex class:

To work with a `Regex` class object, create an instance of the `Regex` class using the desired pattern as the constructor parameter.

[VB]

```
Dim reg As New Regex("\w+(\s+|$)")
```

[C#]

```
Regex reg = new Regex(@"\w+(\s+|$)");
```

This operation retrieves a single match (the first one found)

[VB]

```
Debug.Write(reg.Match("This is a line of text"))
```

[C#]

```
Debug.WriteLine(reg.Match("This is a line of text"));
```

You saw earlier how you can retrieve all of the matches of a string using the Matches property of the Regex class. You can also use the match method to search through the line programmatically as shown here:

[VB]

```
' Here's how to scan through a line
Debug.WriteLine("Scanning through a string")
Dim m As Match
m = reg.Match("This is a line of text")
Do While m.Success
    Debug.WriteLine(m)
    m = m.NextMatch()
Loop
```

[C#]

```
// Here's how to scan through a line
Debug.WriteLine("Scanning through a string");
Match m;
m = reg.Match("This is a line of text");
while (m.Success)
{
    Debug.WriteLine(m);
    m = m.NextMatch();
}
```

Using Static Regex methods

The static Regex methods are very much like the instance methods, except that they include a pattern for the parameter². For example, the first match of a string can be found as follows:

[VB]

```
Debug.WriteLine(Regex.Match("This is a line of text", "\w+(\s+|$)"))
```

[C#]

```
Debug.WriteLine(Regex.Match("This is a line of text", @"\w+(\s+|$)"));
```

You'll want to use the programmatic approach in cases where you're not sure you'll need all of the matches in the string, or where you're only interested in the first match (or are sure that there will be only one match). Regular Expression processing (like any string computational task) takes time, and there's no reason to find matches if you aren't going to use them.

² Regex static methods in C# are called as *Regex.method*. In VB .NET, static methods (called Shared methods) can be called using the Regex class name, or can be invoked from an instance variable. In other words – when working with an instance of the Regex class, VB .NET programmers can call both instance and static methods.

Regex Class Options

The Regex class supports a number of options that modify the behavior of the Regular Expression engine. These options can be set in three ways. When creating an instance of the Regex class, you can pass a RegexOptions enumeration as a constructor parameter. When using a static Regex method, you can choose an override that includes a RegexOptions enumeration parameter. In both cases, the parameter consists of a bit-wise Or of the RegexOptions enumeration values you wish to set. Finally, you can modify the option settings within a group using the syntax

```
(?options-negateoptions:)
```

where *options* consist of one or more of the letters **i m n s** or **x** indicating the option to enable or disable (the meaning of these letters follows).

All Regex class options are off by default.

Ignore Case Option

Set using the RegexOptions.IgnoreCase enumeration value.

Ignore case within a group with:

```
(?i: )
```

Turn on case sensitivity with a group with:

```
(?-i: )
```

SingleLine and MultiLine Options

Set using the RegexOptions.SingleLine and RegexOptions.MultiLine enumeration values.

Set SingleLine or MultiLine mode within a group with:

```
(?s: ) or (?m: ) or both with (?sm: )
```

Turn off SingleLine or MultiLine mode within a group with:

```
(?-s: ) or (?-m: ) or both with (?-sm: )
```

At which point you are probably wondering, how can you set both SingleLine and MultiLine modes at the same time? Aren't they mutually exclusive?

Well, no.

Frankly, this is a rather poor choice of option name. It is much better to think instead of the actual impact of these options on the behavior of the Regular Expression engine.

First, think of SingleLine mode as “Period matches anything” mode.

By default, the `.` pattern matches any character except for the newline character. Thus, if you have the input text:

```
One Two
Three
```

and apply the pattern:

```
One.*
```

The result is:

```
One Two|3
```

However, if you apply the pattern:

```
(?s:One.*)
```

You'll see the following result

```
One Two||Three||
```

The vertical bars represent the `\r` and `\n` character, which is now matched by the period. Why do they call it SingleLine mode? Because from the perspective of the period, the entire text is treated as a single line (i.e. – the `\n` newline character is treated like any other character).

The MultiLine option represents an equally poor choice of name. Think of it as “^ and \$ see lines” mode.

By default, the `^` character matches the start of text, and the `$` matches the end of the text. Looking again at the input text:

```
One Two
Three
```

The pattern:

```
^\w+
```

will by default result in:

```
One
```

This pattern matches the beginning of text followed by one or more word characters (letters, digits and underscore).

Now try the following pattern that turns on “Multiline” mode:

```
(?m:^\w+)
```

The result is:

```
One
Three
```

The `^` pattern character now represents the beginning of each line instead of the beginning of the complete input text.

This leaves us with four possible permutations: You can leave both SingleLine and MultiLine mode off (the default), turn on SingleLine mode, turn on MultiLine mode, or turn on both SingleLine and MultiLine mode (odd though that sounds).

Now, let us get practical for a moment.

You'll find that it is possible to create some very complex Regular Expressions that can do almost everything but wash your dishes. But those can be very difficult to create, understand and support.

In practice, you're mostly going to deal with either single lines of text, or a buffer consisting of multiple lines of text, where you will not want to allow matches to cross a line. This leads to the following conclusions:

³ You'll see an additional vertical bar after the Two if you try this using the RegexIntro's Parse_User menu command. That's because text removed from a text box includes the `\r` (carriage return) as well as `\n` (new line) characters.

- SingleLine mode (which explicitly allows the period pattern to cross lines) is one you'll rarely use.
- MultiLine mode is somewhat more useful when working with multiple lines of text. It's easy enough to match the end of a line (just match the `\n` character), but MultiLine mode changes the `^` pattern character to match the start of each line – which can be extremely useful in making sure that your pattern always starts at the beginning of a line.

ExplicitCapture Option

Set using the `RegexOptions.ExplicitCapture` enumeration value.

Turn on the `ExplicitCapture` option within a group with:

```
(?n: )
```

Turn off the `RegexOptions.ExplicitCapture` option with a group with:

```
(?-n: )
```

This is a tricky one to explain, because it requires that you understand more about capturing than has been discussed up until now. So rather than confuse you, allow me to defer explanation of this option until the section on “Groups and Captures” that follows.

IgnorePatternWhitespace Option

Set using the `RegexOptions.IgnorePatternWhitespace` enumeration value.

Turn on the `IgnorePatternWhitespace` option within a group with:

```
(?x: )
```

Turn off the `IgnorePatternWhitespace` option within a group with:

```
(?-x: )
```

Regular Expression patterns can quickly become very cryptic. That's because everything in the pattern has an impact on the pattern. Patterns can't even cross lines without impacting the meaning of the pattern.

Clearly, this approach is not suitable for complex patterns. Ideally you would like the ability to create multiline Regular Expression patterns using any text editor, even including comments as needed.

The `IgnorePatternWhitespace` option makes this possible.

When you set this option, all white space within the pattern (except for white space within a character class – `[ ]`) is not included in the pattern. That means you can use it to make your pattern readable. You can also use the `#` character to indicate comments (everything after `#` is a comment).

The `mnuIgnorePattern_Click` method in the `RegexIntro` sample project illustrates this. It builds a pattern on the fly, however, you can certainly use a text editor to define patterns using this approach.

[VB]

```
Private Sub mnuIgnorePattern_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles mnuIgnorePattern.Click
```

```

Dim sb As New System.Text.StringBuilder()
sb.Append("(?x:" & ControlChars.CrLf)
sb.Append("# Here is a regular expression" & ControlChars.CrLf)
sb.Append("\w# You can add a comment until the end of the line" _
& ControlChars.CrLf)
sb.Append("+")
sb.Append(Regex.Escape(" "))
sb.Append("\s*|$")
sb.Append(")")
MessageBox.Show(sb.ToString, "Pattern is", MessageBoxButtons.OK)
ParseText(sb.ToString, Nothing)
End Sub

```

[C#]

```

private void mnuIgnorePatternWhitespace_Click(object sender,
System.EventArgs e)
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    sb.Append("(?x:\n");
    sb.Append("# Here is a regular expression\n");
    sb.Append(
        "\w# You can add a comment until the end of the line\n");
    sb.Append("+");
    sb.Append(Regex.Escape(" "));
    sb.Append(@"\s*|$");
    sb.Append(")");
    Clipboard.SetDataObject(sb.ToString());
    MessageBox.Show(sb.ToString(), "Pattern is",
        MessageBoxButtons.OK);
    ParseText(sb.ToString(), null);
}

```

The resulting pattern, as displayed in the message box is as follows:

```

(?x:
# Here is a regular expression
\w# You can add a comment until the end of the line
+\ \s*|$)

```

This pattern is similar to the usual pattern we've been using to extract words from a string, except that it has an extra space. As a result, it won't find the last word in a sentence (unless you add a space after it), but it does illustrate an important point.

The pattern:

```
\w+ \s|$
```

won't work in `IgnorePatternWhitespace` mode, because the space between the `+` and the `\` will be ignored. This means that any time you use the `IgnorePatternWhitespace` option, you must escape all white space characters using the `\` escape character. You must also escape the `#` symbol.

The `Regex.Escape` method can be used to find the escape character for any white space (or other) character as shown in the sample code.

RightToLeft Option

Set using the `RegexOptions.ExplicitCapture` enumeration value.

This option cannot be set within a group.

This option changes the matching direction from right to left.

With this option set, applying the pattern:

```
\w+(\s+|$)
```

to the input string

```
One two three
```

will result in

```
three  
two  
One
```

Note, this reverses the direction of the scan, but the match values themselves remain from left to right. In other words: the characters in the strings are not themselves reversed. This option is probably intended primarily for use with languages that read from right to left.

ECMAScript Option

Set using the `RegexOptions.ECMAScript` enumeration value.

This option cannot be set within a group.

This option can only be used in conjunction with the `MultiLine`, `IgnoreCase` and `Compiled` options.

Earlier in this document I mentioned that Regular Expressions implementations are not standardized. True, you will tend to see common elements in different implementations. You'll probably never see an implementation that doesn't use the period to match any character, or the `?`, `+` and `*` quantifiers to indicate zero or one, one or more, or zero or more of an element. But beyond those common elements, most implementations are unique. Most text editors, for example, including the one built into Visual Studio, provide Regular Expression support for Find and Search & Replace operations. Yet few of these implementations include even close to all of the features provided by the .NET Regular Expression implementation.

There is an organization called ECMA (which was originally an acronym for European Computer Manufacturers Association), whose focus nowadays is developing and sponsoring standards for communications and software. The C# language, for example, has been submitted to ECMA as a proposed standard.

ECMA standard ECMA-262 defines the ECMAScript scripting language, includes the specification for a standard Regular Expression implementation. When you select the ECMAScript option, the .NET Regex object changes its behavior to correspond to the ECMAScript standard. Refer to the ECMA script document (<http://www.ecma.ch>) and MSDN .NET documentation for specifics on these behavior changes.

It is my expectation that the vast majority of .NET programmers will not use this option, if only because, while ECMA does specify a standard, this particular standard's value is

primarily in the area of web page scripting – not general programming where the Regex namespace tends to be used.

Compiled Option

Set using the RegexOptions.Compiled enumeration value.

This option cannot be set within a group.

This option will be discussed in the section titled “Compiling Regular Expressions” in the Additional Topics section of this Ebook.

Groups and Captures

Before continuing, let’s quickly review some of the concepts that you’ve learned so far.

- You’ve learned that the Regex class can apply a Regular Expression pattern to some input text and find “matches” – portions of the input text that match the pattern.
- You know that it is possible to retrieve all of the matches for a pattern in a single operation.
- You know that the pattern can define groups, and that the information captured into a group can be retrieved separately.

This relationship can be seen in the organization of objects in the System.Text.RegularExpressions namespace.

- The Regex object performs the pattern matching operation.
 - A MatchCollection object (containing a collection of Match objects) can be retrieved using the Regex Matches method.
 - Each Match object in the collection describes a single match of the pattern against the input stream.
 - A GroupCollection object (containing a collection of Group objects) can be retrieved from the Match object using its Groups property.
 - Each Group object in the collection describes one of the groups in the match (remember, group zero consists of the entire text of the match).
 - A CaptureCollection object (containing a collection of Capture objects) can be retrieved from the Group object using its Captures property.
 - Each Capture object defines the captured text for the specified group.
 - A CaptureCollection object can also be retrieved from the Match object using its Captures property.

This hierarchy indicates how you use the objects in the namespace. From an internal implementation point of view, the Group object inherits from the Capture object, and the Match object inherits from the Group object.

The Capture object defines the following three properties:

- Index The index in the input string of the first character of this capture
- Length The length of this capture
- Value The string data of this capture

A Group object adds the following properties:

- **Captures** The CaptureCollection object containing any Captures by this group.
- **Success** True if at least one Capture was made by the group.

How is it possible for a Group object to have no Captures? The existence of a group in a match depends on the original pattern. A match can succeed even if some of the groups do not capture any data. For example: The pattern (A)|(B) will match the letters “A” or “B”. Both groups will exist in the match, but only one will have captured data!

The Match object adds the Groups property that allows you to retrieve the groups for the match.

I realize that this can be quite confusing – yet it is important to understand it. In fact, understanding how text is captured into groups forms the heart of using Regular Expressions effectively.

The best way to learn Regular Expressions is through experimentation. The RegexTester sample program is a useful tool for performing that experimentation.

The RegexTester example

The RegexTester program screen is shown in Figure 1.

Regular Expression patterns are added to the Pattern text box. The input string text box can be edited directly, or loaded from a file. The TreeView window displays two types of data. First comes a list of Matches, each of which contains the Groups and then the Captures for those groups in a hierarchy. Next comes the same list of Matches, followed by the Captures for each match (as retrieved from the Captures property).

The Tools-Parse menu command is used to execute the Regular Expression search on the input string. The Tools menu also includes a SimpleTest menu, whose event code contains some short code fragments that you’ve already seen in this ebook (specifically, the examples shown under creating and using a Regex class, and using Regex static class methods).

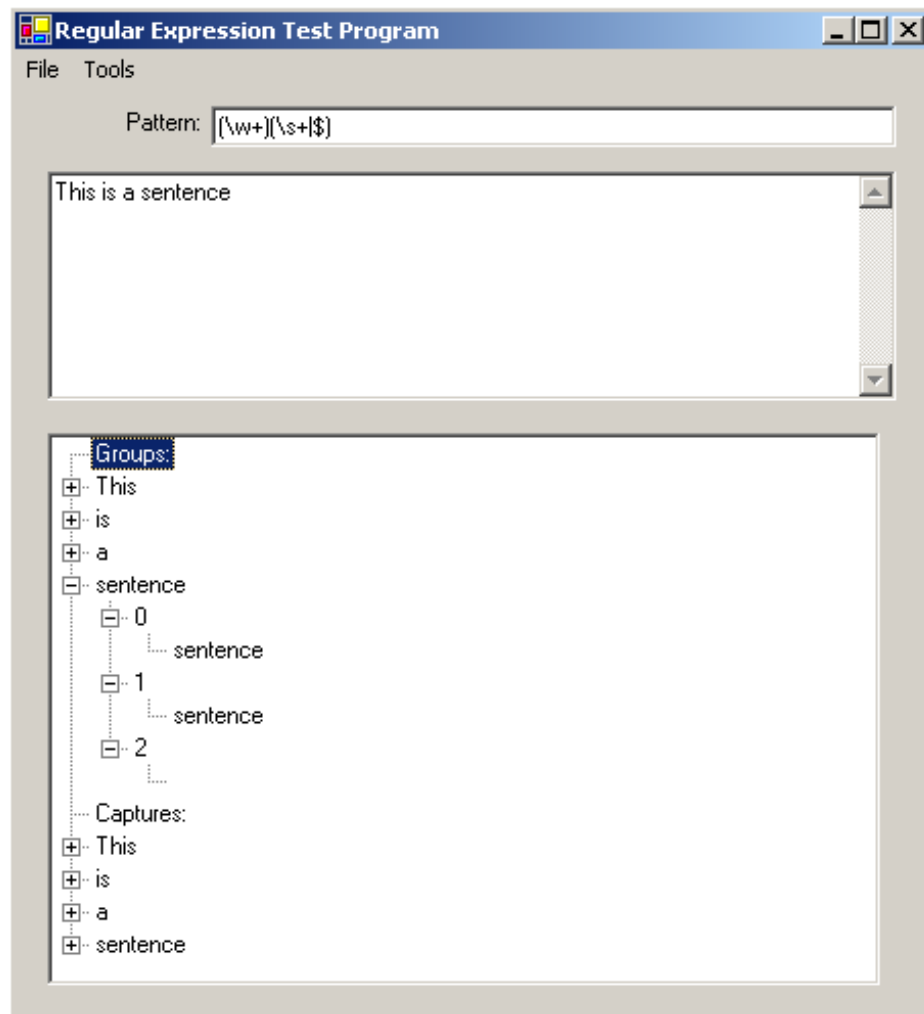


Figure 1 – Main window of the RegexTester application

The real work of the program is accomplished in the ParseTheString function that is shown here:

[VB]

```
Private Sub ParseTheString()
    Dim rx As New Regex(txtPattern.Text)
    Dim mc As MatchCollection
    Dim m As Match
    Dim GroupNumbers() As Integer
    Dim GroupNameIndex As Integer
    ' Perform the Regex Match
    mc = rx.Matches(txtInput.Text)
    ' Clear existing nodes and add the Groups heading
    tvResult.Nodes.Clear()
    tvResult.Nodes.Add("Groups:")
    ' GroupNumbers is an array that contains the numbers
    ' of any group that also has a name
    GroupNumbers = rx.GetGroupNumbers()
```

```

For Each m In mc
    ' For each match, add a list of groups
    Dim gps As GroupCollection
    Dim gp As Group
    Dim GroupNumber As Integer
    Dim tvmatch As New TreeNode(m.Value)
    tvResult.Nodes.Add(tvmatch)
    gps = m.Groups
    For GroupNumber = 0 To gps.Count - 1
        ' We don't use For...Each here because we
        ' need the group number
        gp = gps(GroupNumber)
        ' See if this group number is present
        ' in the GroupNumbers array which means it has a name
        GroupNameIndex = Array.IndexOf(GroupNumbers, GroupNumber)
        Dim tvgroup As TreeNode
        If GroupNameIndex >= 0 Then
            ' It has a name, display the
            ' name instead of group number
            tvgroup = New _
                TreeNode(rx.GroupNameFromNumber(GroupNameIndex))
        Else
            ' Unnamed group, display the number
            tvgroup = New TreeNode(GroupNumber.ToString())
        End If
        tvmatch.Nodes.Add(tvgroup)
        Dim cps As CaptureCollection
        Dim cp As Capture
        cps = gp.Captures
        ' For each group, add a list of captures
        For Each cp In cps
            Dim tvcapture As New TreeNode(cp.Value)
            tvgroup.Nodes.Add(tvcapture)
        Next
    Next
Next
Next

' Similar to what was shown above, but without the groups
tvResult.Nodes.Add("Captures:")
For Each m In mc
    Dim cps As CaptureCollection
    Dim cp As Capture
    Dim tvmatch As New TreeNode(m.Value)
    tvResult.Nodes.Add(tvmatch)
    cps = m.Captures
    For Each cp In cps
        Dim tvcapture As New TreeNode(cp.Value)
        tvMatch.Nodes.Add(tvcapture)
    Next
Next
End Sub

```

[C#]

```

private void ParseTheString()
{
    Regex rx = new Regex(txtPattern.Text);
    MatchCollection mc;
    int[] GroupNumbers;
    int GroupNameIndex;
    // Perform the Regex Match
    mc = rx.Matches(txtInput.Text);
    // Clear existing nodes and add the Groups heading
    tvResult.Nodes.Clear();
    tvResult.Nodes.Add("Groups:");
    // GroupNumbers is an array that contains the numbers
    // of any group that also has a name
    GroupNumbers = rx.GetGroupNumbers();
    foreach (Match m in mc)
    {
        // For each match, add a list of groups
        GroupCollection gps;
        Group gp;
        int GroupNumber;
        TreeNode tvmatch = new TreeNode(m.Value);
        tvResult.Nodes.Add(tvmatch);
        gps = m.Groups;
        for (GroupNumber = 0; GroupNumber <= gps.Count - 1;
            GroupNumber++)
        {
            // We don't use For...Each here because we need the
            // group number
            gp = gps[GroupNumber];
            // See if this group number is present in the
            // GroupNumbers array which means it has a name
            GroupNameIndex = Array.IndexOf(GroupNumbers, GroupNumber);
            TreeNode tvgroup;
            if (GroupNameIndex >= 0 )
            {
                // It has a name, display the name instead
                // of group number
                tvgroup = new
                    TreeNode(rx.GroupNameFromNumber (GroupNameIndex));
            }
            else
            {
                // Unnamed group, display the number
                tvgroup = new TreeNode (GroupNumber.ToString());
            }
            tvmatch.Nodes.Add(tvgroup);
            CaptureCollection cps;
            cps = gp.Captures;
            // For each group, add a list of captures
            foreach (Capture cp in cps)
            {
                TreeNode tvcapture = new TreeNode (cp.Value);
                tvgroup.Nodes.Add(tvcapture);
            }
        }
    }
}

```

```

    }
    // Similar to what was shown above, but without the groups
    tvResult.Nodes.Add("Captures:");
    foreach(Match m in mc)
    {
        CaptureCollection cps;
        TreeNode tvmatch = new TreeNode(m.Value);
        tvResult.Nodes.Add(tvmatch);
        cps = m.Captures;
        foreach(Capture cp in cps)
        {
            TreeNode tvcapture = new TreeNode(cp.Value);
            tvmatch.Nodes.Add(tvcapture);
        }
    }
}

```

You'll find that the `RegexTester` program is an excellent tool for understanding and experimenting with Regular Expressions.

Groups in Depth

You've read about group hierarchies and how the various `System.Text.RegularExpressions` namespace objects relate to each other, both from a functional and inheritance perspective. You've seen code that illustrates how the key properties for these objects are used.

But there's a good chance you still don't really understand it – not intuitively. My goal now is to fix that.

You'll do most of your real learning by experimenting on your own, but let's start by doing some together.

Let's start by applying a variation of the familiar word break pattern:

```
\w+ (\W+ | $)
```

This expression breaks down as follows:

- `\w+` Matches one or more “word” characters (letters, digits and underscore).
- `(` Opens an unnamed group
- `\W+` One or more non-word characters (includes punctuation marks and white space).
- `|$` or the end of the line
- `)` Closes the group

When applied to the sentence

```
This is, a sentence!
```

the `RegexTester` program produces the following results in the Groups section:

```

Groups:
This
  0
    This
  1

is,
  0
    is,
  1
    ,
a
  0
    a
  1

sentence!
  0
    sentence!
  1
    !

```

As you can see, each match has two group. Group zero consists of the entire match (and in fact, it's value is the same as that of the match. Group 1 consists of everything that appears between the words.

Now, in a practical case, you're probably more interested in the words than in the punctuation between the words. So you would most likely place the word capture into a group as follows:

```
(\w+) (\W+|$)
```

This pattern results in the following RegexTester output:

```

Groups:
This
  0
    This
  1
    This
  2

is,
  0
    is,
  1
    is
  2
    ,
a
  0
    a
  1
    a
  2

```

```

sentence!
  0
    sentence!
  1
    sentence
  2
    !

```

As you can see, group 1 now contains the word and group 2 contains the word separators. You might be wondering about why you would need group 2 in the first place. The parenthesis for the separators exists to help readability regarding the precedence, not to capture the information. You can improve the performance and efficiency of the Regular Expression processor by not capturing groups that you don't need. This can be accomplished in two ways.

You can use the non-capturing group syntax:

```
(?: )
```

Or you can set the explicit capture option mentioned earlier, either setting the `RegexOptions.ExplicitCapture` option, or using the syntax:

```
(?n: )
```

When you set the `ExplicitCapture` option, the only groups that capture data are those that are explicitly named using the syntax:

```
(?<name>)
```

It's important to understand that whether a group captures or not has no impact on the regular expression match (other than the obvious fact that you can't use backreferencing to reference a group that doesn't capture). It just makes the group effectively disappear – it is not assigned a number, it does not show up in the collection of groups for the match.

Change the word break pattern to the following: `(\w+) (?:\W+|$)`

```

Groups:
This
  0
    This
  1
    This
is,
  0
    is,
  1
    is
a
  0
    a
  1
    a
sentence!
  0
    sentence!
  1
    sentence

```

As you can see, group #2 is gone.

The ExplicitCapture option can accomplish the same task. Consider this pattern:

```
(?n: (?<words>\w+) (\W+|$))
```

This expression breaks down as follows:

(?n:	Turn on the explicit capture option. Only capture explicitly named groups:
(?<words>	Open a group named “words”
\w+)	Capture one or more “word” characters
(\W+ \$)	Capture one or more non-word characters, or the end of the line
)	Close group 0.

This results in the following RegexTester output:

```
Groups:
This
  0
    This
  words
    This
is,
  0
    is,
  words
    is
a
  0
    a
  words
    a
sentence!
  0
    sentence!
  words
    sentence
```

As you can see, other than group zero (which represents the entire match), only the named group “words” captures data.

Captures in Depth

So far you’ve only seen the case where each group has a single capture. You might be wondering how it’s possible for a group to have more than one capture. Let’s say by accident instead of using the code

```
(\w+) (\W+|$)
```

for word breaks, you used the following:

```
(\w) + (\W+|$)
```

The resulting RegexTester output will now be:

```

Groups:
This
  0
    This
  1
    T
    h
    i
    s
  2

is,
  0
    is,
  1
    i
    s
  2
    ,

a
  0
    a
  1
    a
  2

sentence!
  0
    sentence!
  1
    s
    e
    n
    t
    e
    n
    c
    e
  2
    !

```

Now you'll note that the matches themselves are unchanged. That's because

`(\w+)` - a group consisting of one or more word characters
will match the same text as

`(\w)+` - one or more repetitions of a group containing one word character
See the difference?

When you have repetitions of a group, each repetition will have its own captured data.

Let's consider now a simple CSV format example consisting of three fields separated by commas:

```
First field,Second field,Third field
```

You saw how to split this string using the pattern `,\s*` in the section on “Splitting A String” earlier in this ebook. Here’s one pattern that can be used to parse this string:

```
(.+?) (?:,\s*|)$)
```

This expression breaks down as follows:

- `(.+?)` Find one or more characters. Because the comma character is matched by a period, capture the smallest possible number of characters needed to make the match (try on your own to see what happens if you leave the `?` out of this expression).
- `(?:` Start a non-capturing group
- `,\s*` Match a comma, followed by any number of white space characters.
- `|$)` or match the end of line and close the group.

The RegexpTester program produces the following results:

```
First field,
0
    First field,
1
    First field
Second field,
0
    Second field,
1
    Second field
Third field
0
    Third field
1
    Third field
```

As you can see, `field` is a separate match.

Now consider the following pattern:

```
((.+?) (?:,\s*|$))+
```

This is almost identical to the previous pattern. All we’ve done is place the entire pattern in a group and specify that a match consists of one or more of that group.

Now the results are very different:

```
Groups:
First field,Second field,    Third field
0
    First field,Second field,    Third field
1
    First field,
    Second field,
    Third field
2
    First field
    Second field
    Third field
```

What has changed?

First, you now have a single match. This makes sense, because a match now consists of one or more strings that match the CSV field. In this case the match consists of all three fields.

Group #0, as usual, consists of the entire match.

Group #1 is the group consisting of the `((.+?)(?:,\s*|$))` term. You've captured three instances of this group, as you can see in the output.

Group #2 is the group consisting of `(.+?)`. This group also has three instances, but in this case only the field itself is captured – the separators were matched, but not captured by the non-capturing group `(?:,\s*|$)`.

Why would you take this approach instead of the first one? The most likely case is if you were trying to extract one or more CSV fields that were part of a larger pattern.

In most cases, choice of grouping is much more important than the use of captures within a group – since groups can be accessed in the regular expression (for backreferencing, or as elements in a search and replace operation), whereas captures of groups are only useful when you are processing the results of a Regular Expression operation using software.

You might be wondering, what happens if you use backreferencing or reference a group (in a search and replace operation) if the group has multiple captures?

In that case, the value of the group will be that of the most recent capture.

Part III - Advanced Regular Expressions

Up until now you've seen the most of the escapes and grouping constructs that you will be using with Regular Expressions. In this section, I'll introduce some of the more advanced escapes and pattern constructs and show you how they can be used. Unless otherwise noted, all of the tests in this section are done in the RegexTester program.

Zero-Width Assertions

A zero width assertion is one that does not capture any information. It's called an assertion because it "asserts" something about the way the pattern is to be processed.

\A, \Z and \z

You've already seen the ^ and \$ characters that match the start or end of a string or line (depending on the use of the RegexOptions.Multiline option). The \A and \z assertions match the start and end of the input string respectively. The \Z escape is like \z except that if there is a newline \n at the end of the string it will not be matched. Unlike The ^ and \$ assertions, \A, \Z and \z are not influenced by the Multiline option.

As with many constructs, this is best seen through example.

Enter the following lines into the input text box in the RegexTester program (be sure to enter an extra Return after the second line – the third line is empty).

```
String with Newline First
String with Newline Second
```

The resulting string will consist of two lines of data, each followed by a carriage return line feed (CR LF or \r\n) pair. Let's look at the results of several different patterns:

```
\w+$
```

This pattern captures nothing. Why? Because \$ matches the end of the string or before the \n at the end of the string. There is a CR \r between the last character and the \n. You can match with:

```
\w+\r$
```

This pattern captures the word "Second" along with the \r character. Now look what happens when you turn on multiline mode:

```
(?m:\w+\r$)
```

This pattern results in two matches: "First\r" and "Second\r" (in both cases it captures the \r character as well). Now consider the \Z escape:

```
(?m:\w+\r\Z)
```

This captures the word "Second" along with the \r character. It is just like \$ without multiline mode (multiline mode has no impact on \Z).

If you try \z with the following pattern:

```
(?m:\w+\r\z)
```

you will get no matches. Why? Because \z matches the very end of the line, unlike \Z which matches the end of the line or before a newline if one exists. Change the pattern to:

```
(?m:\w+\r\n\Z)
```

This captures the word “Second” along with the \r and \n characters.

\b and \B

The \b assert matches a word boundary (specifically, a transition from a \w word character to a \W non-word character, or vice versa). \B matches a non-word boundary.

For example: the following pattern:

```
(?i:A\w*)
```

Finds every place where the letter a appears (in upper or lower case), and captures all of the word characters that follow. When applied to the sentence:

```
Alpha Beta Gama
```

This captures:

```
Alpha
a
ama
```

Which is not particularly useful.

Add \b to the beginning as follows:

```
(?i:\bA\w*)
```

To capture only the word Alpha. The A is only matched if it follows a word boundary.

\G

This is rather subtle assertion that asserts the beginning of the location from which the Regular Expression engine is searching.

If you are just scanning through a string, by using the Regex.Matches property or using Match.NextMatch in a loop, this assert has little use, since each match automatically begins at the end of the previous match.

However, if you are manually selecting the start location in the input string, \G asserts that the match begin at the start location you specified.

The RegexAdvanced project includes function SlashG that is defined as follows:

[VB]

```
Public Shared Sub SlashG()
    Dim inputstring As String = "A1234,B1234,CA134,A1234,C1234,A1234"
    Dim m As Match
    Dim c As Integer
    Dim r As New Regex("\GA.*?(,|$)")
    For c = 0 To 5
        ' Match override sets location of start of search
        m = r.Match(inputstring, c * 6)
        If m.Success Then Console.WriteLine(m.Value)
    Next
End Sub
```

[C#]

```

public static void SlashG()
{
    string inputstring = "A1234,B1234,CA134,A1234,C1234,A1234";
    Match m;
    int c;
    Regex r = new Regex(@"\GA.*?(,|$)");
    for (c = 0; c < 6; c++)
    {
        // Match override sets location of start of search
        m = r.Match(inputstring, c * 6);
        if ( m.Success )
        {
            Console.WriteLine(m.Value);
        }
    }
}

```

The inputstring variable contains six fields, each 6 characters long (including the comma). The goal here is to match every field that begins with an A. You might start with the pattern:

A.*?(,|\$)

This expression breaks down as follows:

A	The letter A
.*	Zero or more characters (any character except \n) – matching as few as possible.
(, \$)	A comma or the end of the input text.

This results in the following matches:

```

A1234,
A134,
A134,
A1234,
A1234
A1234

```

The first search returns the first field. The next two match the A134 part of the third field. This field doesn't begin with an A, and it is found twice, but that is correct because the patterns does nothing to force the match to start at the start location.

Your first thought might be to use the ^ assert to match the start of the string, changing the pattern to:

^A.*?(,|\$)

But this results in only:

```

A1234,

```

That's right. The ^ assert matches the start of the input string – not the location from which you start searching! When you use \G at the start of the string

\GA.*?(,|\$)

You get these results:

```
A1234,
A1234,
A1234
```

Which corresponds to every field starting with the letter A.

Zero Width Pattern Assertions

The following four patterns look like groups, but they're actually assertions. They allow you to determine if an arbitrary pattern precedes or follows the current location in the search, but they don't match the pattern. The patterns are:

- `(?=pattern)` Asserts that the specified pattern follows this location. Does not backtrack (see explanation in the section titled "Non-backtracking Constructs").
- `(?!pattern)` Asserts that the specified pattern does not follow this location. Does not backtrack (see explanation in the section titled "Non-backtracking Constructs").
- `(?<=pattern)` Asserts that the specified pattern precedes this location. Does not backtrack (see explanation in the section titled "Non-backtracking Constructs").
- `(?<!pattern)` Asserts that the specified pattern does not precede this location. Does not backtrack (see explanation in the section titled "Non-backtracking Constructs").

Let's say you are searching text for last names that appear in the form Mr. *name*, Mrs. *name*, or Ms. *name*. Consider the text string:

```
Mr. Jones, Mrs. Smith and Ms. Gates all went to the conference.
```

You could use the following pattern:

```
((Mr.)|(Mrs.)|(Ms.))\s+(?<name>\w+)
```

This expression breaks down as follows:

- | | |
|-----------------------------|--|
| <code>(</code> | Opens a group that will match Mr., Mrs. or Ms. |
| <code>(Mr.)</code> | Matches Mr. |
| <code> (Mrs.)</code> | Matches Mrs. |
| <code> (Ms.)</code> | Matches Ms. |
| <code>)</code> | Closes the group that matches the honorific. |
| <code>\s+</code> | Matches one or more spaces. |
| <code>(?<name></code> | Opens a group named "name". |
| <code>\w+</code> | Matches one or more word characters. |
| <code>)</code> | Closes group "name". |

This pattern will match the full name including the honorific. You can extract the last name from the "name" group. But you can also use an assertion to match only the last name. Consider this pattern:

```
(?<=((Mr.)|(Mrs.)|(Ms.))\s+)(?<name>\w+)
```

This expression breaks down as follows:

(?<=	Opens a lookbehind assertion
((Mr.)(Mrs.)(Ms.))	Group that matches Mr., Mrs. or Ms.
\s+	Matches one or more spaces
)	Closes the lookbehind assertion.
(?<name>\w+)	Matches one or more word characters and captures them into group “name”.

Let’s take a close look at the results of the RegexTester program:

```

Jones
0
    Jones
1
    Mr.
2
    Mr.
3
4
name
    Jones
Smith
0
    Smith
1
    Mrs.
2
3
    Mrs.
4
name
    Smith
Gates
0
    Gates
1
    Ms.
2
3
4
    Ms.
name
    Gates

```

These results can be a bit confusing. The actual matches are Jones, Smith and Gates, as you would expect. Group #1 is the group that contains groups #2, #3 and #4 (Mr., Mrs. and Ms.). As you can see, Group #1 matches and captures the honorific. However, because group #1 is enclosed within a zero-width assertion, the data is not captured into the match itself!

You can simplify matters by using non-capturing groups as follows:

```
(?<=(?: (?:Mr.) | (?:Mrs.) | (?:Ms.)) \s+) (?<name>\w+)
```

Which results in:

```

Jones
0
    Jones
name
    Jones
Smith
0
    Smith
name
    Smith
Gates
0
    Gates
name
    Gates

```

More on Quantifiers

You've already seen that quantifiers are used heavily in Regular Expressions. The `?`, `*` and `+` quantifiers are the most common. Appending a `?` to each of these quantifiers (`??`, `*?` and `+`?) specifies that you wish to capture as few characters as possible. You may also see the term “greedy” and “non-greedy” to describe this phenomena. A quantifier such as `*` that captures as many characters as possible is called “greedy”. A quantifier such as `*?` that captures as few characters as possible is “non-greedy.”

You can also specify an exact number or range to capture using the following quantifiers:

- `{n}` Repeat exactly *n* times.
- `{n,}` Repeat at least *n* times, matching as many times as possible.
- `{n,}??` Repeat at least *n* times, matching as few times as possible.
- `{n,m}` Repeat at least *n* times, but no more than *m* times.
- `{n,m}??` Repeat at least *n* times, but no more than *m* times, matching as few times as possible.

Consider the case where you wish to extract zip codes from addresses that are formatted like this one:

```

Joe Smith
12345 Someplace Ln.
San Jose, CA 95131-2345

```

Let's start with a pattern that matches zip codes:

```
\d{5}(-\d{4})?
```

This expression breaks down as follows:

- `\d` Matches a digit.
- `{5}` Exactly five repetitions
- `(` Start a group.
- `-` Match the `-` character.
- `\d{4}` Matches exactly four digits.
- `)` Close the group.

? Matches zero or one repetitions of the group (i.e. `-\d{4}` is optional).

This results in the following matches:

```
12345
95131-2345
```

The address number is captured as well because it just happens to be five digits long. There are a number of ways to eliminate the address number. In this format, the address number always appears at the start of a line. In multiline mode, you could use the `^` character to indicate the start of a line. However, you can match the start of any line except the first just as easy using the term `[\n]`. This defines a character set that matches any character except for the newline character. Since the address number in this format follows the newline character, it will not match the following expression:

```
[\n]\d{5}(-\d{4})?
```

Try this with five digit zip-codes as well.

More on Grouping

You've already learned the most important grouping constructs, but there are a few more powerful constructs that you should be aware of.

Balancing Group Definitions

Regular Expressions can perform surprisingly sophisticated tasks. For example: let's say you had a math expression such as:

```
1 + ((5 + 3) + 8 + (4 + (7))) + 6
```

How would you go about obtaining a list of every set of data that is within parenthesis. In other words, you're looking for the results:

```
(5 + 3) + 8 + (4 + (7))
5 + 3
4 + (7)
7
```

In order to do this, you must be able to not just find a right paren – you have to be able to find the correct right paren – the one that matches any given left paren! This task applies to any scenario where you have the possibility of nested data (nested parenthesis, nested brackets, nested If...Then statements, nested XML tags, etc.).

You would most likely tackle this by using two Regular Expressions. Because we are looking for the matching right parent for each left paren, we first have to find the location of each left parent. That will define the start location for each of the four searches. The `RegexAdvanced` sample includes function `CommaMatching` that performs this task as shown here:

[VB]

```

Public Shared Sub CommaMatching()
    Dim inputstring As String = "1 + ((5 + 3)+ 8 + (4 + (7))) + 6"
    Dim parenmatches As MatchCollection
    Dim rxparens As New Regex(inputstring)
    ' First find the locations of left parens
    parenmatches = rxparens.Matches(inputstring, "\(")

    Dim rx As New Regex("\(.*\)")

    Dim parenmatch As Match
    Dim m As Match
    Dim gps As GroupCollection
    For Each parenmatch In parenmatches
        ' Perform the match for each left paren found
        m = rx.Match(inputstring, parenmatch.Index)
        If m.Success Then
            gps = m.Groups
            Console.WriteLine(m.Value)
            Dim x As Integer
            Dim groupnums() As Integer = rx.GetGroupNumbers
            Dim groupnames() As String = rx.GetGroupNames
            For x = 0 To groupnums.Length - 1
                Console.WriteLine("    Group #: "
                    & groupnums(x).ToString & " name: " &
                    groupnames(x) & " value = " & gps(x).Value)
            Next
        End If
    Next
End Sub

```

[C#]

```

public static void CommaMatching()
{
    string inputstring = "1 + ((5 + 3)+ 8 + (4 + (7))) + 6";
    MatchCollection parenmatches;
    Regex rxparens = new Regex(inputstring);
    // First find the locations of left parens
    parenmatches = Regex.Matches(inputstring, @"\(");

    Regex rx = new Regex(@"\(.*\)");

    Match m;
    GroupCollection gps;
    foreach(Match parenmatch in parenmatches)
    {
        // Perform the match for each left paren found
        m = rx.Match(inputstring, parenmatch.Index);
        if (m.Success)
        {
            gps = m.Groups;
            Console.WriteLine(m.Value);
            int x;
            int[] groupnums = rx.GetGroupNumbers();

```

```

String[] groupnames = rx.GetGroupNames();
for(x = 0;x<= groupnums.Length - 1;x++)
{
    Console.WriteLine("    Group #: " +
        groupnums[x].ToString() + " name: " +
        groupnames[x] + " value = " + gps[x].Value);
}
}
}

```

The first step is simple. The “(“ pattern matches the left paren character. We could have simply used the `String.IndexOf` method (or VB `Instr` command) with a loop to find all the left parens, but this approach is quite effective and returns a `MatchCollection` where each `Match` object’s `Index` property indicates the location of the paren.

The `rx` `Regex` object is the one that is intended to find the matching right paren. After the `Match` method is called (with the left paren location as the start point for the search), the routine displays the match, and the number, name and value of each group.

One might think to try the following initial pattern:

```
\(.*\)
```

This expression breaks down as follows:

```

\(  
.*  
\)

```

Matches a left paren.
Matches any character except `\n`, matching as many as possible
Matches a right paren

This results in the following output:

```

((5 + 3)+ 8 + (4 + (7)))
    Group #: 0 name: 0 value = ((5 + 3)+ 8 + (4 + (7)))
(5 + 3)+ 8 + (4 + (7)))
    Group #: 0 name: 0 value = (5 + 3)+ 8 + (4 + (7)))
(4 + (7)))
    Group #: 0 name: 0 value = (4 + (7)))
(7)))
    Group #: 0 name: 0 value = (7)))

```

This pattern always finds the last right paren. Obviously not what we’re looking for.

To understand how to find a matching paren, you need to understand two constructs. The first one is:

```
(?<thisgroup-othergroup>pattern)
```

This is a balancing group definition. It’s a bit tricky to understand (took me several hours to puzzle it out from Microsoft’s documentation), so I’ll go slowly.

You know that normally, every time the pattern defined in a group generates a match, that match captures data. If a group matches multiple times (through the use of a quantifier), each match creates a new capture for the group. The “capture” value for a group is the most recent capture (so if you obtain the `Value` for a `Group` object, you get the value of the most recent `Capture` for the group).

When the pattern for a balancing group definition generates a match, the data is not captured into the group. Instead, the Regular Expression engine checks to see if

othergroup (the name of a different group) has a value. If it does, all of the data between group *othergroup* and the current match are stored in *thisgroup*. The most recent capture for *othergroup* is deleted. Thus, if *othergroup* had three captures before applying this construct, it would have two afterwards.

Now consider this pattern:

```
(?: (?<ltparen>\( ) | (?<rtparen-ltparen>\) ) | [^\)\(] ) *
```

This expression breaks down as follows:

(?:	Open a non-capturing group
(?<ltparen>	Open a group named ltparen)
\(	Matches a left paren
)	Close group ltparen. This group matches the left paren character.
	or match the following
(?<rtparen-ltparen>	Open a balancing group named rtparen, that balances group ltparen
\)	Matches a right paren
)	Close group rtparen
	or match the following
[^\)\(]	Matches any character except for a left or right paren
)	Close the non-capturing group
*	Accept zero or more of the non capturing group

In plain English, this pattern matches zero or more characters, where each character is either a left paren (matched by group ltparen), a right paren (matched by group rtparen), or something else (matched by character class [^\)\(]).

Let's walk through what happens when this pattern is applied to the input string (remember, we're starting the search from the position of the first left paren):

```
((5 + 3) + 8 + (4 + (7))) + 6
```

<u>Input</u>	<u>What happens</u>
(	Matches group ltparen. The left paren becomes the first capture and current value of the group.
(	Matches group ltparen. The left paren becomes the second capture and current value of the group. The first capture is still stored as a capture for this group.
5+3	Match the [^\)\(] character set because they aren't parens.
)	Matches group rtparen. rtparen captures everything between group ltparen and this match – which will be 5+3 in this case. The current capture for ltparen is thrown away, leaving the first capture whose value is a left paren.
+ 8 +	Match the [^\)\(] character set because they aren't parens.

- (Matches group ltparen. This left paren becomes the new second capture and current value of the group. The first capture is still stored as a capture for this group.
- 4 + Match the `[^\\)\(]` character set because they aren't parens.
- (Matches group ltparen. This left paren becomes the third capture and current value of the group. The first and second captures are still stored as captures for this group.
- 7 Matches the `[^\\)\(]` character set because it isn't a paren.
-) Matches group rtparen. rtparen captures everything between group ltparen and this match – which will be 7 in this case. The current capture for ltparen is thrown away, leaving the second capture whose value is a left paren (the one before 4).
-) Matches group rtparen. rtparen captures everything between group ltparen and this match – which will be 4 + (7) in this case. The current capture for ltparen is thrown away, leaving the first capture whose value is a left paren (the one at the start of the string).
-) Matches group rtparen. rtparen captures everything between group ltparen and this match – which will be (5 + 3)+ 8 + (4 + (7)) in this case. The current capture for ltparen is thrown away, leaving no captures in group ltparen.
- + 6 Match the `[^\\)\(]` character set because they aren't parens.

Group 2 now contains the text between the first parenthesis and its matching right paren!

The results for each left paren are as follows:

```
((5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 0 name: 0 value = ((5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = (5 + 3)+ 8 + (4 + (7))
(5 + 3)+ 8 + (4 + (7))
  Group #: 0 name: 0 value = (5 + 3)+ 8 + (4 + (7))
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 4 + (7)
(4 + (7))
  Group #: 0 name: 0 value = (4 + (7))
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 4 + (7)
(7)
  Group #: 0 name: 0 value = (7)
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 7
```

Wait a minute! What happened to the second paren? It found 4 + (7) in group rtparen. Why is this?

Because there is no mechanism in this pattern to stop after you've found a matching paren. The Regular Expression engine will continue to scan, finding further left parens and their matching right parens. In effect, this pattern doesn't find the matching right paren for the first left paren. It finds the matching left paren for the last right paren!

In order to stop we need to use the alternating construct:

```
(?(groupname)yes|no)
```

This construct works as follows:

If the group specified in *groupname* has a value (i.e. it has already successfully matched), apply the pattern that appears in the *yes* part of the group, otherwise apply the pattern that appears in the *no* part of the group.

We can use this to solve our parenthesis problem by recognizing that the *ltparen* group has no captures only at two times – at the very start, and immediately after the matching paren is found for the first left paren!

This leads us to the following expression:

```
(?: (?(ltparen>\() | (?(rtparen-ltparen>\)) | (?(ltparen) [^\(\)]*|.*)) ) *
```

This expression breaks down as follows:

(?:	Open a non-capturing group
(?(ltparen>	Open a group named ltparen)
\(	Matches a left paren
)	Close group ltparen. This group matches the left paren character.
	or match the following
(?(rtparen-ltparen>	Open a balancing group named rtparen, that balances group ltparen
\)	Matches a right paren
)	Close group rtparen
	or match the following
(?(ltparen)	Start an alternating yes/no group based on the current value of the ltparen group
[^\(\)]*	If ltparen has a value, we are still looking for the matching right paren. This pattern matches as many characters as it can up to the next left or right paren
.*	If ltparen has no value, we have just found the matching right paren. So perform a greedy capture of all the rest of the characters in the string immediately – capturing any more parenthesis as well!
)	Close the alternating yes/no group
)	Close the non-capturing group
*	Accept zero or more of the non capturing group

In this expression, rather than capturing non-paren characters one at a time, we make a choice. If the matching right paren has not yet been found, the expression captures characters up until the next paren. If the matching right paren has been found, the expression captures the rest of the line immediately. If there are any more left/right paren pairs, it won't matter, since they'll be captured by this part of the expression.

The results are as follows:

```

((5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 0 name: 0 value = ((5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = (5 + 3)+ 8 + (4 + (7))
(5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 0 name: 0 value = (5 + 3)+ 8 + (4 + (7))) + 6
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 5 + 3
(4 + (7))) + 6
  Group #: 0 name: 0 value = (4 + (7))) + 6
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 4 + (7)
(7))) + 6
  Group #: 0 name: 0 value = (7))) + 6
  Group #: 1 name: ltparen value =
  Group #: 2 name: rtparen value = 7

```

As you can see, The rtparen has successfully captured the content of each matching pair of parenthesis in the input text.

I encourage you to walk through the expression character by character as I did for the earlier example if you are still confused. Regular Expressions are not particularly intuitive, and I assure you that it took me substantial trial and error to develop this example.

Non-Backtracking Constructs

Let's take a closer look at what happens when you apply a pattern that contains alternation or wildcards. Consider the following pattern:

```
(?s:<tag>(.*?)</tag>)
```

This expression breaks down as follows:

(?s:	Sets single line option (so the period will match newline characters.
<tag>	Matches the text "<tag>"
(.*)	Matches any character (including newline). Captures the text into group #1.
</tag>	Matches the text "</tag>"

When applied to the input text:

```

<tag>
  Some text
</tag>

```

This will capture the entire text, placing the line "Some text" along with surrounding white space and control characters into group #1.

Now, here's the question:

You know that the pattern `(.*)` matches all characters. Why did the Regular Expression engine not capture the third line `</tag>` into this group and simply not find a match? After all, the `*` quantifier is supposed to capture as many characters as possible. It should have just gone on and captured all of the remaining text.

The reason this didn't happen is because the .NET Regular Expression engine is able to backtrack – it can go back and reevaluate a pattern when multiple choices exist. For example: the first time through you might see:

```
Pattern element: <tag>  (.* )                </tag>
Captures:      <tag>    Some text </tag>
Result:  ok   ok   fails   No match
```

So the Regular Expression Engine might try again, backtracking by seeing what happens if it doesn't take capture all of the characters possible into the .* term.

```
Pattern element: <tag>  (.* )                </tag>
Captures:      <tag>    Some text </      tag>
Result:  ok   ok   fails   No match
```

Finally it might backtrack far enough to see a match.

```
Pattern element: <tag>  (.* )                </tag>
Captures:      <tag>    Some text          </tag>
Result:  ok   ok   ok      Match found!
```

Backtracking can occur any time a quantifier or alternation occurs.

The (?>) construct disables backtracking within a group. Thus if you use the pattern:

```
(?s:<tag>(?!>.* )</tag>)
```

This changes the (.*) group to be non-backtracking. This means that it will capture as many characters as possible and the search will continue from that point. Since it captures the </tag> term, no match will result for the input string.

Now consider this pattern:

```
(?s:<tag>(?!>([<]*) )</tag>)
```

The (?>.*) term has been replaced by the (?>[<]*) term which breaks down as follows:

```
(?>      This is a non-backtracking non-capturing group
(        Opens group #1
[<]     Matches any character that is not a < character.
*        Matches zero or more times.
)        Closes group #1
```

This pattern works!

It works because there is no need for backtracking – the (?>[<]*) term stops capturing as soon as it sees the < character, at which point the </tag> term captures the rest of the string.

Why are non-backtracking groups important?

Because backtracking is slow. In a complex expression, it can be very slow.

It's important to keep in mind that zero-width pattern assertions (described earlier) are always non-backtracking. So, if they contain any capturing groups, those groups will be non-backtracking as well.

For example: Imagine a major database accident has occurred, causing all of the text fields to be merged together into one line, resulting in lines such as:

```
95125CaliforniaInvoice103925
```

You might think to use the following pattern to extract the information:

```
(\d+) (?= ([A-Za-z]+) )\2 (\d+)
```

This expression breaks down as follows:

(\d+)	Matches one or more digits into group #1.
(?=	Lookahead assertion.
([A-Za-z]+)	Matches one or more letters into group #2.
\2	Matches group #2
(\d+)	Matches one or more digits into group #3.

This will result in the following:

```
95125CaliforniaInvoice103925
0
    95125CaliforniaInvoice103925
1
    95125
2
    CaliforniaInvoice
3
    103925
```

Now let's say you wanted to exclude the word Invoice from the match. You might try this pattern:

```
(\d+) (?= ([A-Za-z]+) )\2Invoice (\d+)
```

Logically, this would work. The group in the Lookahead assert might capture "CaliforniaInvoice" at first, but then would backtrack to capture just "California" in order to allow an overall match. However, this would involve backtracking, and zero-length pattern assertions are non-backtracking constructs.

I must confess this particular example is a bit artificial – there are simpler patterns that would serve this purpose, but it does illustrate the point.

Advanced Search and Replace

You learned earlier that the .NET Regular Expression implementation includes a powerful search and replace capability. This capability is made all the more powerful through the use of programmatic substitutions. It is possible to specify a MatchEvaluator delegate as a parameter to the Regex Replace method. This delegate points to a method that takes the current match as a parameter and return the value that should be used to replace the entire match in the input string. The RegexAdvanced sample program contains the following sample code that demonstrates how this works.

[VB]

```
Public Shared Sub SearchAndReplace()
    Dim inputstring As String = _
        "Dear Mr. {1}, you owe us a total of ${2}. Pay it now!"
    Dim replacestring As String
    replacestring = Regex.Replace(inputstring, "{(\d+)}", _
        AddressOf Evaluator)
```

```

        Console.WriteLine(replacestring)
    End Sub

    Public Shared Function Evaluator(ByVal m As Match) As String
        Dim matchval As Integer
        matchval = CInt(m.Groups.Item(1).Value)
        Select Case matchval
            Case 1 ' Get the name here - perhaps from a database?
                Return ("Jones")
            Case 2 ' Get the amount owed here
                Return ("100,000")
        End Select
    End Function

```

[C#]

```

public static void SearchAndReplace()
{
    string inputstring =
        "Dear Mr. {1}, you owe us a total of ${2}. Pay it now!";
    string replacestring ;
    replacestring = Regex.Replace(inputstring, @"{(\d+)}",
        new MatchEvaluator(Evaluator));
    Console.WriteLine(replacestring);
}

public static string Evaluator(Match m)
{
    int matchval;
    matchval = Convert.ToInt32(m.Groups[1].Value);
    switch(matchval)
    {
        case 1: // Get the name here - perhaps from a database?
            return ("Jones");
        case 2: // Get the amount owed here
            return ("100,000");
    }
    return(null);
}

```

The resulting string is:

```

    Dear Mr. Jones, you owe us a total of $100,000. Pay it now!"

```

Part IV - Additional Topics

Compiling Regular Expressions

The `Regex` class can be created using the `RegexOptions.Compiled` enumeration flag. Normally, when you create an instance of a `Regex` class, the Regular Expression pattern is compiled internally into internal instructions for processing the Regular Expression. These instructions are (presumably) optimized for this purpose.

When you select the `RegexOptions.Compiled` flag, the pattern is compiled into .NET Intermediate Language (IL) code. This code is in turn compiled by the JIT compiler into native code. The code created using this option is loaded into the current `AppDomain` and is not unloaded until the `AppDomain` is terminated. So this approach should only be used for patterns that are used frequently (so as not to clutter up memory with lots of compiled Regular Expression code).

According to the documentation, compiled Regular Expressions should perform better than those that are not. Frankly, I have not been able to successfully verify this in my own benchmarking. The Regular Expression engine does keep a cache of those patterns compiled into internal instructions as well, so if you create an instance of a `Regex` object and use it multiple times, the performance according to my measurements to date, will be similar to the performance of a `Regex` object compiled into IL code.

If you are defining a library of Regular Expression patterns, you can provide a reusable library of `Regex` objects by compiling all of the `Regex` classes you'll be using into a separate assembly. The compilation will be done at the assembly build time.

The `RegexTester` sample program shows how to create these assemblies. Use the Tools-Compile menu command and enter the desired type name for the new `Regex` object.

[VB]

```
Private Sub mnuCompile_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles mnuCompile.Click
    Dim typename As String
    typename = InputBox("Enter type name")
    If typename = "" Then Exit Sub
    ' Each Regex class will form it's own type in the new assembly
    Dim compileinfo() As RegexCompilationInfo = {New _
        RegexCompilationInfo(txtInput.Text, RegexOptions.Compiled, _
        typename, "CompiledRegex", True)}

    ' This example builds a weak named assembly
    ' Set the other AssemblyName fields for strong naming
    Dim aname As AssemblyName = New System.Reflection.AssemblyName()
    aname.Name = "CompiledRegex"

    ' This builds the assembly .DLL file
    Regex.CompileToAssembly(compileinfo, aname)
End Sub
```

[C#]

```
private void mnuCompile_Click(object sender, System.EventArgs e)
{
    string typename;
    frmInputBox ibox = new frmInputBox();
    ibox.ShowDialog();
    typename = ibox.textBox1.Text;
    ibox.Dispose();
    if (typename == null) return;
    // Each Regex class will form it's own type in the new assembly
    RegexCompilationInfo[] compileinfo = {new
    RegexCompilationInfo(txtInput.Text, RegexOptions.Compiled,
    typename, "CompiledRegex", true)};

    // This example builds a weak named assembly
    // Set the other AssemblyName fields for strong naming
    AssemblyName aname = new System.Reflection.AssemblyName();
    aname.Name = "CompiledRegex";

    // This builds the assembly .DLL file
    Regex.CompileToAssembly(compileinfo, aname);
}
```

In this sample code, `typename` is the name of the type. `CompiledRegex` is the name of both the assembly and the namespace. Each `RegexCompilationInfo` object you provide defines a new type (derived from `Regex`) in the assembly that uses the pattern you specify.

You can add a reference to the `CompiledRegex.dll` assembly from any .NET application, then create a new instance of any of the enclosed `Regex` classes as shown here:

[VB]

```
Dim newregex As New CompiledRegex.Testit()
```

[C#]

```
CompiledRegex.Testitnewregex = new CompiledRegex.Testit();
```

Performance Considerations

Here are some suggestions to improve performance when using the .NET Regular Expression objects.

If you plan to use the same `RegularExpression` pattern multiple times, do not use the static `Regex` methods. Instead, create a `Regex` object with the pattern, and reuse that object. When you create the `Regex` object, the pattern is compiled (either into an internal Regular Expression instruction set or IL code depending on use of the `RegexOptions.Compiled` enumeration flag). When you reuse the `Regex` object, it does not need to repeat this initial compilation step.

Avoid backtracking when possible. Use Lookahead and Lookbehind instead. This is described in the Advanced Regular Expressions section.

If you use alternation, use non-backtracking groups when possible as described in the Advanced Regular Expressions section.

Threading Issues

As with many .NET classes, only the static members of classes should be assumed to be thread safe. This means that different threads can call Regex static members safely at any time. However, once you've created an instance of a Regex class (or are using instances of Group, Match and Capture objects), you should synchronize access to those objects.

Part IV - What are State Machines, and Why Should You Care?

I have intentionally avoided going into details and theory of the .NET Regular Expression engine. Nevertheless, there is value in learning just a bit more – especially if you are not yet familiar with the principles of Finite State Machines. You may find what you learn helps you as much or more than what you have learned about Regular Expressions.

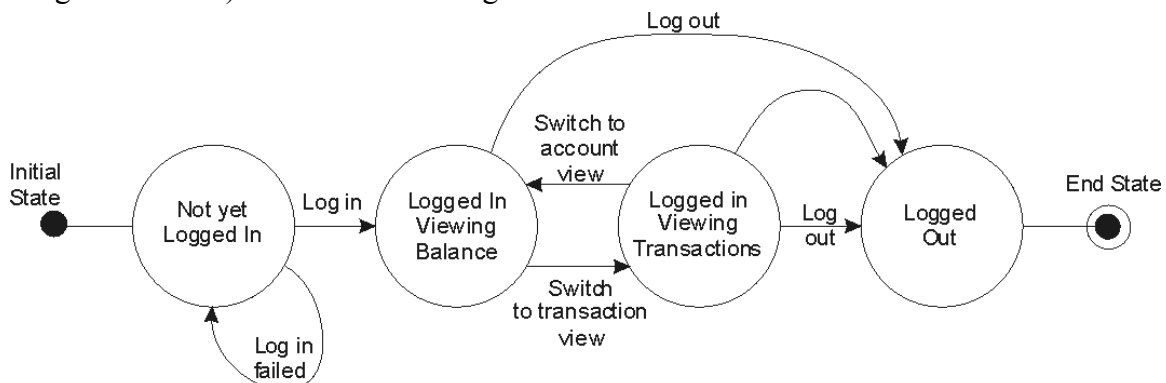
A State Machine, in the context of software, is a way of organizing the operations that take place in an application. The idea is that your program exists in a finite number of possible states, and that something happens to move your program from one state to the next.

For example: You might have a web application that allows you to log in and view your account information. This could be divided into the following states:

- Not yet logged in.
 - Action: Display the login page
- Logged in and viewing the account balance
 - Display the account balance
 - Display logout button, and view transaction button.
- Logged in and viewing recent transactions
 - Display recent transactions
 - Display logout button, and view account balance button.
- Logged out
 - Display “Goodbye page”

At any given time, the application exists in one of these four states. In each state there exist a limited number of possible events (typically called “messages” when discussing state machines). For example, during the “Logged in” state, the page displays text boxes for the user name and password and a “login” button. If the user enters a correct login, the application switches to the “Logged in and viewing the account balance” state. If the login fails, the user sees an error message and the application remains in the “Not logged in state”.

State machines are frequently described using State Diagrams (or State Transition Diagrams – STD) that look something like this:



The initial state is where the state machine starts (the circle marked Initial state, and the one marked End State are not states themselves – rather pointers to the actual initial and end states). Each state is represented by a circle. Each arrow represents a message that can be received by the state machine (often triggered by an event) that can cause the machine to change states. Thus, a successful log-in moves the application into the account viewing state.

Why are State Machines Important?

To understand why state machines are so important, remember that much of what we do as software developers consists of managing complexity. We implement very complex applications and algorithms by breaking them up into smaller manageable tasks.

One of the key purposes of object oriented programs (OOP) is to help manage complexity. By using information hiding within objects (implementing private functions and variables) and defining a limited number of public methods and properties, you are able to deal with an object as a single indivisible entity. Once the object is created and implemented, you no longer have to worry about how it works or the possibility of accidentally modifying one of its internal data variables. Thus Object Oriented Programming inevitably results in simpler programs – programs that are easier to understand and support.

State machines serve the same purpose, but on an architectural level. As part of the development of a state machine you define all of the valid events that may occur during that state. For each event you define an action and a state transition (which includes the possibility of remaining in the same state). You might also define an action for all invalid states.

What does this accomplish?

- First, it makes the program far easier to modify. Adding new features might consist of adding new states. By clearly defining the events that can bring you to that state and events for that state you at the same time define the code that needs to be modified. Code that is not involved with that state can be safely ignored.
- It becomes dramatically easier to test programs implemented as state machines. Why? Because it is possible to break down the testing process into testing of individual states. If you test each state for all of its possible events (a reasonable task), you can go a long way to eliminating bugs in your program. True, there may remain subtle bugs due to the interaction of states (especially if they share any data), and your state machine may itself have design flaws (say, forgetting a particular event), but the results of this approach will always result in a higher quality program than otherwise.
- Using state machines also demands that you spend some time designing them before you start coding. And let's face it, design time is something that developers often skimp on, especially in the face of deadline pressures.

Here's another way of looking at it.

Figure 2 shows how object oriented programming reduces complexity by reducing the number of functions and variables you need to deal with at a given level of your program. In this illustration you can see on the left side a large number of variables and functions

that might appear in a non OOP program. When using OOP the variables and functions are hidden inside of objects. These objects expose a limited number of methods that provide a high level encapsulation of the enclosed variables and functions. As a result, once you've implemented these objects, instead of having to worry about a large number of variables and functions (and their interactions), you need only concern yourself with a small number of objects and their methods. Fewer items to work with results in reduced complexity, increased reliability, and overall lower software development costs.

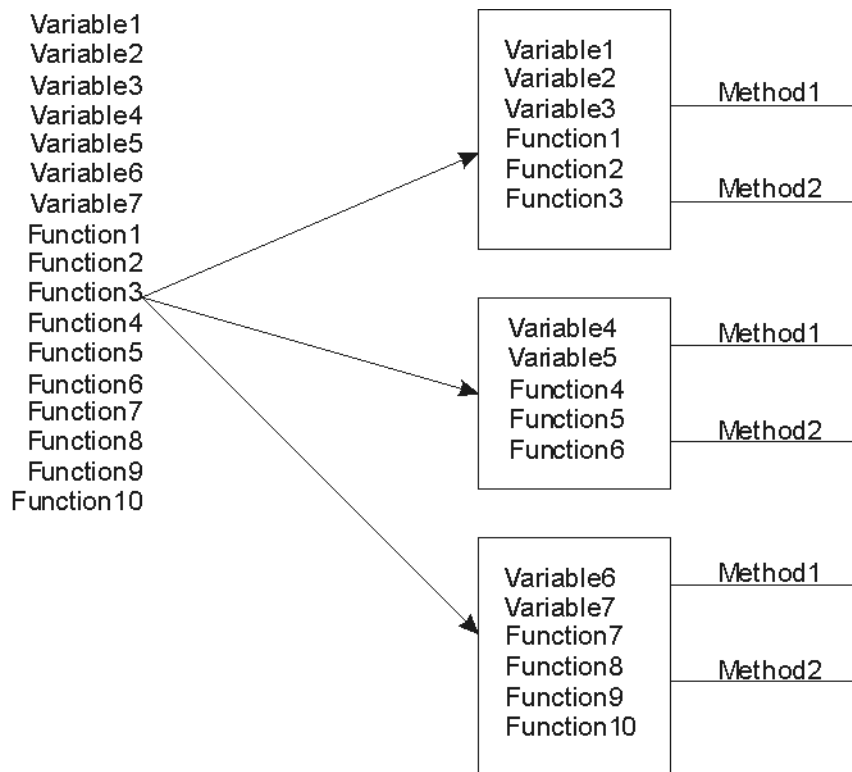


Figure 2 – Object oriented programming.

Now consider figure 3. On the left side, instead of lists of functions and variables you can see a list of events. These are possible inputs to your program. These can be in the form of user actions, data received from a network, data read from a disk or other source and even results of an operation or exceptions that occur while a program is running - basically anything that can represent input to your program.

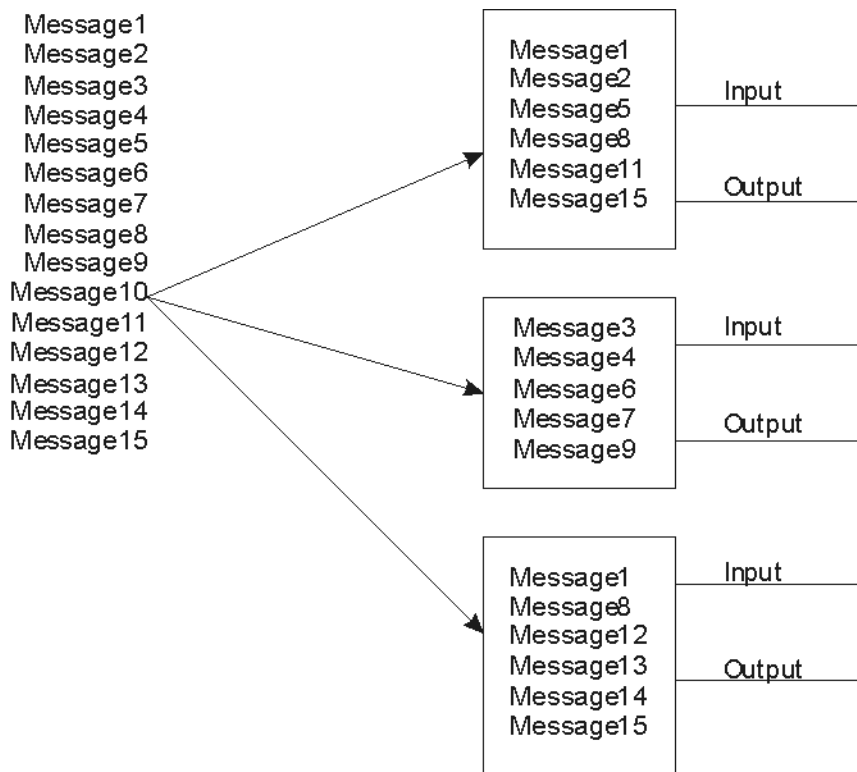
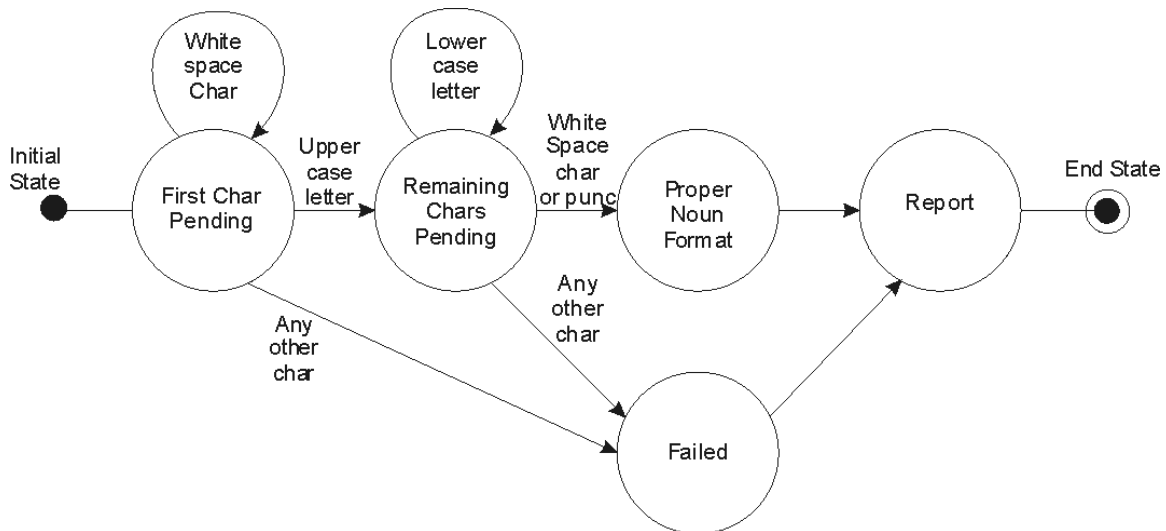


Figure 3 – State machine programming.

If your code has to consider all possible inputs at all times, the complexity of the any non-trivial program would be impossible to deal with. Fortunately, this is rarely the case – programs naturally deal with certain events at certain times. A function that reads data from disk rarely worries about user input. Yet at the same time, that function that reads data from disk may receive unanticipated input – a user abort or disk error, and the failure to deal with unanticipated input is a key source of bugs and instability in software.

A state machine serves to divide an application's life into a series of states, each of which has a set of acceptable input. Once in a given state, you need only concern yourself with valid inputs. Invalid inputs are by definition errors that can be trapped and handled. Just as OOP simplifies a program by reducing the number of variables and functions you need to deal with, state machines simplify a program by reducing the number of inputs you need to deal with. Collapsing a given set of inputs into a state machine that has a set start point and end point, and can be dealt with as a single entity, just as an object can deal with a group of variables and functions as a single entity.

State machines work at multiple levels. Consider the following state machine:



This represents a state machine that processes incoming characters to look for words in the format of a proper noun (i.e., the first character is upper case, all other characters are lower case).

The first state handles three possible messages. A white space character (such as space or tab) indicates the word has not yet started, so the machine remains in the same state. An upper case character means that the word has started, so the machine transitions into the second state. Any other character represents a failure, so the machine moves into the Failed state. Once in the second state, all subsequent lower case letters indicate continuation of the word, so the machine remains in the same state. A white space or legal punctuation indicates the end of the word, which moves the machine into the “success” state. any other character again moves the machine into the failed state.

The state machine you see here implements the Regular Expression pattern:

```
\s*[A-Z][a-z]*
```

Guess what. When you create a Regex object with this pattern, internally it implements a state machine! The process of finding matches and other Regular Expression tasks all involves the execution of a state machine defined using a pattern.

State machines have applicability far beyond the processing of text. They can simplify your code, improve it’s reliability and maintainability, improve thread safety and reduce both development costs and lifecycle costs. Desaware’s latest product for .NET:

Desaware’s StateCoder™ provides a framework that makes it easy to create powerful state machines in .NET. You can read more about it in appendix B and online at <http://www.desaware.com/statecoderL2.htm>.

Part V - Conclusion

Regular Expressions represent a technology that is not necessarily familiar to many Visual Basic or C++ programmers. Even those familiar with Regular Expressions may not be in the habit of using them. This is because until the appearance of .NET, a regular expression library has not been an integral part of either Visual Basic or the standard C++ libraries. Now that Regular Expression technology is included in the .NET framework, it represents a feature of .NET with which every .NET programmer should become familiar.

This concludes my third ebook. It is my belief that ebooks, or edocuments, represent a solution to the rather odd situation that exists today.

If you wish to learn about something (or write about it) you currently have two choices:

- You can look for (or write) articles for magazine or webzines.
- Or you can look for (or write) books.

Articles tend to be relatively short – under 10 pages. Enough to cover a specific tip or technique, but not enough to provide depth or context.

Books typically have to have at least 150 pages to be economical. And there are many subjects that don't need that much coverage (do you really think you need to read another 100 pages to start using Regular Expressions?).

Ebooks are ideal for the 15-100 page length – short enough to print out and read on your own, long enough to provide an in depth treatment of shorter subjects. I hope you agree.

Meanwhile, I thank you for your support in purchasing this book (and if you received it without purchasing it, please use the Amazon honor link at the start of the book to pay for it, the recommended price, or what you think it was worth).

Dan Appleman

dan@desaware.com

February 2002

Index

.NET Framework classes	19
Capture Class	26, 34
Group class.....	26, 31
Regex Class.....	19
Alternating Operators.....	8
Appendix A.....	67, 70
Apress	75
Backreferencing	10, 14
Books	
Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0	74
Dan Appleman's Visual Basic Programmer's Guide to the Win32 API	74
Dan Appleman's Win32 API Puzzle Book & Tutorial for VB Programmers.....	74
eBooks.....	72
Exploring .NET	73
Hijacking .NET	72
How Computer Programming Works	72
Introduction to NT/2000 Security Programming with Visual Basic 6	72, 73
Moving to VB.NET: Strategies, Concepts and Code	72
Regular Expressions with .NET.....	73
Tracing and Logging with .NET.....	73
Visual Basic .NET or C#: Which to Choose?.....	72
Capture class	26
Capture Class	34
Captured values.....	11
CAS/Tester.....	70
Character sets	10
Code download	2
Compiling Regular Expressions	54
Data Validation	18
Desaware	
CAS/Tester.....	70
Desaware Event Log Toolkit.....	71
NT Services Toolkit.....	71
SpyWorks.....	70
StateCoder.....	61, 70
StorageTools	71
VersionStamper.....	71
Desaware ActiveX Gallimaufry.....	71
Escapes.....	7
Considerations for C#	8
Event Log Toolkit.....	71
Example Programs	
RegexAdvanced	39, 44, 52
RegexIntro.....	4, 9, 12, 23

RegexTester	27, 38, 54
FTP site	2
Group class.....	26, 31
Grouping	
advanced	44
grouping and backreferences	10, 68
non-backtracking.....	50
Match	7, 12
NT Services Toolkit.....	71
Operations	12
Parsing HTML	12
Parsing text.....	3
Patterns.....	6
Performance Considerations	55
Programs	
RegexAdvanced	39, 44, 52
RegexIntro.....	4, 9, 12, 23
RegexTester	27, 38, 54
Publishing	75
Quantifiers.....	8, 43
Reference	
Regular Expression Pattern.....	67
alternating constructs	69
assertions.....	68
comments	69
grouping and backreferences	68
quantifiers	69
replacement text.....	69
single character escapes	67
Regex Class.....	19
creating.....	19
Options.....	21
Compiled.....	26
ECMAScript	25
ExplicitCapture	23
Ignore Case	21
IgnorePatternWhitespace	23
RightToLeft.....	25
SingleLine and MultiLine	21, 38
Regex methods.....	20
RegexAdvanced	39, 44, 52
RegexIntro.....	4, 9, 12, 23
RegexOptions.Compiled flag.....	54
RegexTester	27, 38, 54
Regular Expression Pattern Reference.....	67
alternating constructs	69

assertions	68
comments	69
grouping and backreferences	68
quantifiers	69
replacement text	69
single character escapes	67
Regular Expressions	
backreferencing	10
captured values	11
character sets	10
compiler	3
compiling	54
data validation	18
escapes	7
grouping and backreferences	10
interpreter	3
match	7, 12
objects	19
operations	12
Options	21
patterns	6, 7
processor	3
search and replace	14
special characters	7
substrings	16
Regular Expressions - Advanced	38
Grouping	44
balancing group definitions	44
non-backtracking constructs	50
quantifiers	43
search and replace	52
Zero-Width Assertions	38
\A, \Z and \z	38
\b and \B	39
\G	39
Zero-Width Pattern Assertions	41
Role based security	72
Sample code	2
Search and Replace	14, 52
Security	
Role based	72
Special characters	7
Splitting a String	16
SpyWorks	70
State Machines	57
StateCoder	61, 70

StorageTools	71
Substrings.....	16
System.String.Replace method	15
System.String.Split method	17
Threading Issues	56
Validation.....	18
VersionStamper.....	71

Appendix A – Regular Expression Pattern Reference

Single Character Escapes

The following list shows the .NET Regular Expression escape characters that match single characters.

- `\a` matches ASCII character 7
- `\b` matches ASCII character 8 (backspace) if inside a set of characters `[ ]` or in a replacement string. Otherwise represents a word boundary.
- `\d` matches any decimal digit `[0-9]`
- `\D` matches any character that is not a decimal digit `[^0-9]`
- `\e` matches the escape character ASCII character `&H1B` or `0x1B`
- `\f` matches ASCII character `&HC` or `0xC`
- `\n` matches a newline (LF) character (ASCII character `&H0A` or `0x0A`)
- `\r` matches a return (CR) character (ASCII character `&H0D` or `0x0D`)
- `\s` matches any white spaces (space, tab, newline, `\f`, `\v`, `\r`)
- `\S` matches any character that is not a white space.
- `\t` matches a tab character
- `\v` matches ASCII character `&HB` or `0xB`
- `\w` matches a character (a-z, A-Z, 0-9 and underscore)
- `\W` matches any character that is not a letter.
- `\Onnn` matches an octal character code with value `nnn` (`n` is a digit 0-8). It must have a leading zero. And you'll never use it because nobody uses octal anymore.
- `\xnn` matches a hex character code with value `0xnn` or `&Hnn`.
- `\cC` matches a control character, such as Control-C or Control-D
- `\n` matches the previous group. `n` is one or more digits not starting with zero. See Grouping and Backreferencing for more information.
- `\unnnn` matches a hexadecimal Unicode character with value `0xnxxx` or `&Hnnnn`
- `\` in front of any of the characters `. $ ^ { [ ( | ) * + ? \` match the character itself. Thus `\(` matches a left paren, and not the opening of a group.
- `.` matches any character other than the newline character (or any character in single-line mode)
- `[cccc]` matches any character `c` found between the `[]` brackets. Thus `[abcd]` would match `a`, `b`, `c` or `d`. Character ranges (`0-9A-Za-z`) may be specified
- `[^cccc]` matches any character other than those found after the `^` character. Thus `[^abcd]` would match any character other than `a`, `b`, `c` and `d`.

Assertions

These “zero-width” assertions specify a location in the input string, but do not capture data.

<code>^</code>	Asserts the start of the input string, or start of line (in multiline mode)
<code>\$</code>	Asserts the end of the input string, or end of line (in multiline mode) – before <code>\n</code> if present.
<code>\A</code>	Asserts the start of the input string.
<code>\Z</code>	Asserts the end of the input string (before <code>\n</code> if present).
<code>\z</code>	Asserts the end of the input string.
<code>\b</code>	Asserts a boundary between word and non-word characters (a word character is one that matches <code>\w</code>).
<code>\B</code>	Asserts a location that is not a boundary between word and not-word characters.
<code>\G</code>	Asserts the beginning location of the current search.
<code>(?= pattern)</code>	Asserts that the specified pattern follows this location. Does not backtrack (see explanation for <code>(?>)</code>).
<code>(?!pattern)</code>	Asserts that the specified pattern does not follow this location. Does not backtrack (see explanation for <code>(?>)</code>).
<code>(?<=pattern)</code>	Asserts that the specified pattern precedes this location. Does not backtrack (see explanation for <code>(?>)</code>).
<code>(?<!pattern)</code>	Asserts that the specified pattern does not precede this location. Does not backtrack (see explanation for <code>(?>)</code>).

Grouping and Backreferences

<code>(pattern)</code>	Defines a group. The pattern within the parenthesis defines the data captured for the group. Groups are numbered in order of left parenthesis starting with one (group zero represents the entire match). Note: For the remainder of the definitions, <i>pattern</i> is implied. Thus <code>(?:)</code> is the same as <code>(?:pattern)</code>
<code>(?:)</code>	Non capturing group. The group is not assign a name or number and will not appear in the Groups collection for the Match.
<code>(?>)</code>	Non backtracking group. The group captures as much text as possible based on the pattern in one pass.
<code>(?<thisgroup-othergroup>)</code> or <code>(?'thisgroup-othergroup')</code>	Refer to the explanation in the “Balancing Group Definitions” section under Advanced Regular Expressions.
<code>(?options-negateoptions:)</code>	Enables or disables one or more of the specified <i>options</i> which consist of one or more of the letters i m n s or x . See the section on Regex Class Options for details.

Quantifiers and Alternating constructs

You can append the following special characters to indicate a repetition of the previous character or group.

*	Repeat zero or more times matching as many times as possible.
+	Repeat one or more times matching as many times as possible.
?	Repeat zero or one time matching as many times as possible.
??	Repeat zero times or one time, matching zero if possible.
*?	Repeat zero or more times matching as few times as possible.
+?	Repeat one or more times matching as few times as possible.
{n}	Repeat exactly n times.
{n,}	Repeat at least n times, matching as many times as possible.
{n,}?	Repeat at least n times, matching as few times as possible.
{n,m}	Repeat at least n times, but no more than m times.
{n,m}?	Repeat at least n times, but no more than m times, matching as few times as possible.
	When between two characters or groups, matches one or the other.
(?(<i>pattern</i>) <i>yes</i> <i>no</i>)	The Regular Expression pattern specifies is applied at the current point in the search. The expression does not capture (equivalent to (?! <i>pattern</i>)). If the expression would match, the <i>yes</i> pattern is applied, otherwise the <i>no</i> pattern is applied. The expression may include a backreference group specifier, in which case it behaves like the following construct.
(?(<i>groupname</i>) <i>yes</i> <i>no</i>)	If the specified <i>groupname</i> currently is matched (i.e. has captured data), the <i>yes</i> pattern is applied, otherwise the <i>no</i> pattern is applied.

Replacement Text

Replacement text can include the following special characters.

\$n	Insert group #n at this point.
\${name}	Insert the group named “name” at this point.
\$\$	Insert the \$ character at this point.

Comments

The following comment constructs can be used in Regular Expressions..

(?#comment)	Everything from the # to the right paren is a comment and plays no role in the matching.
# comment	When the IgnorePatternWhitespace option is set (See RegexOptions), everything after the # symbol to the end of the current line is a comment.

Appendix B - Books and Products by Dan Appleman

As an author, and CEO of Desaware, I'd like to take this opportunity to tell you about my other books, Desaware's products, and Apress, a leading edge publisher that I co-founded.

I'd also like to invite you to visit our Web site at: www.desaware.com for detailed product and book descriptions, product demos, FAQ pages and additional technical articles.

Software

CAS/TesterTM

CAS/Tester is an automated Code Access Security tester for .NET assemblies. Because you can never be certain what permissions are allowed on a target system, it is essential that you test your assemblies under a variety of configurations and make sure it will fail gracefully regardless of how a system is configured. CAS/Tester executes your assembly on multiple configurations (over 80 predefined, and no limit to the number you can define). Detailed reports show exactly what exceptions occur under each scenario and where.

For components, class libraries, Windows Forms controls, Windows Forms applications and console applications – CAS/Tester will help your developers implement code access security, and testers verify behavior.

StateCoderTM

StateCoder is a .NET namespace that is designed to make it easy to create and support powerful State Machines in .NET using Visual Basic .NET, C# and other .NET Languages.

With Desaware's StateCoder, you will create .NET code that is more reliable, easier and cheaper to test, support, understand and modify.

With sophisticated thread management, it is ideal for creating multithreaded applications and component, including asynchronous design patterns, background operations, and protocols.

SpyWorks Professional and Standard Editions

You *can* do that in Visual Basic—*SpyWorks* is the tool that lets you do things in VB, VB.Net and C# that are normally not possible.

- Use hooks to detect messages or keystrokes for selected windows or the entire system.
- Export functions from Visual Basic DLL's, **C# or VB.NET DLL assemblies!**
- Use advanced subclassing techniques including the ability to subclass other applications.

- COM edition supports a wide variety of sophisticated operations, allowing you to do virtually anything in VB6 that is possible using VC++.

And much more—Visit our Web site for details.

NT Services Toolkit

Available in both COM and .NET editions.

The .NET edition provides numerous features beyond those provided by the service framework included with .NET. Among these are the ability to easily expose objects simultaneously via DCOM and .NET Remoting, the ability to test the service without actually installing it as a service, and self-installation features.

The COM edition allows you to create powerful services with VB6 – you can even run services and debug them from within the VB6 environment.

Desaware Event Log Toolkit

Desaware's *Event Log Toolkit* makes creation of custom event sources easy, and provides all the tools needed to create and log custom events. Both VB6 and .NET developers will benefit from the ability to precisely define event log messages, severity and categories. Includes VB6 classes (with source) for advanced event log management and reporting.

VersionStamper

Safely distribute your COM component-based applications, avoiding “DLL-Hell”. VersionStamper verifies that applications have the correct versions of DLL, OCX and other components, offering remote diagnostics, reporting and the ability to make your applications and components self-updating.

VersionStamper has saved our customers fortunes in support costs. Download our demo and find out for yourself how they did it.

StorageTools

OLE Structured Storage is the technology Microsoft uses with their own office applications to store multiple streams of complex data into a single file. Easier to use than a database, and more flexible for hierarchical data storage, StorageTools is the key to using structured storage from within Visual Basic, C#, VB .NET and other .NET languages.

The Desaware ActiveX Gallimaufry

This eclectic set of ActiveX controls written in Visual Basic is both useful and educational. Includes full VB source code. These include: a taskbar control, common dialog controls, TWAIN control (for scanners and digital cameras) and more. Includes full VB6 source.

[Books by Dan Appleman](#)

[Moving to VB.NET: Strategies, Concepts and Code, 2nd ed.](#)

Written for Visual Basic 6 developers who are ready to learn and migrate to .NET, this book is ideal to not only get you started, but give you the foundation you need to progress further.

In Strategies you'll learn when and why to migrate to .NET, and when you shouldn't. In Concepts you'll not only learn new concepts, but will unlearn old design patterns that can get you into trouble. And in Code you'll learn about the changes to the language itself, along with a thorough introduction to the .NET framework, including key concepts such as distributed programming and security.

[How Computer Programming Works](#)

This fully illustrated beginner's book is the perfect book for friends, family and kids – anyone who would like to know more about programming. It's the book to read before you get a programming book, teaching key concepts like variables, compilers, program flow, etc. Think of it as a computer science course for everyone.

[eBooks by Dan Appleman](#)

[Hijacking .NET](#)

Each eBook in this series explores ways you can make use of undocumented or hidden capabilities within the .NET framework.

- Volume 1 discusses role based security, showing how to use private functions to enumerate roles for an account and set security for files and directories.

[Visual Basic .NET or C#: Which to Choose?](#)

In this best-selling E-book, you will find an in-depth comparison of the two languages. In a feature by feature, head to head contest, you'll learn there really is a best choice, but that it depends on your specific situation.

[Obfuscating .NET: Protecting your code from prying eyes](#)

Did you know that you ship your complete source code any time you distribute a .NET assembly?

One of the consequences of the architecture of .NET is that a great deal of information about an assembly is kept with the assembly in a part of the file called the Manifest. This information makes it remarkably easy to not just recompile the assembly, but to decompile it, make modifications, then recompile it. In this PDF-eBook, you'll learn about a technique called *Obfuscation*, that can help you avoid this problem. And you'll receive an in depth look at one particular approach to obfuscating your .NET assemblies, **along with a link to a free download of Desaware's open source QND-Obfuscator.**

Regular Expressions with .NET

It might surprise you to know that yet another language is built into Visual Studio – one that can be used in conjunction with VB .NET or any other .NET language. Regular Expressions is an extremely powerful language designed to parse and manipulate blocks of text. Ideal for processing text and validation, understanding Regular Expressions is essential for any .NET developer.

This eBook is intended to be a complete introduction to Regular Expressions that can even be read and understood by programmers who have never heard of them. It is also intended to help experienced Regular Expression programmers come up to speed quickly on the .NET implementation of Regular Expressions.

Tracing and Logging with .NET

The .NET framework includes a powerful tracing and diagnostic system, one that goes far beyond the simple Debug.WriteLine most developers are accustomed to. In this eBook will introduce you to the .NET diagnostic system, teach you tracing and logging design patterns, and demonstrate advanced techniques such as tracing directly into a database or directing trace data directly to an outgoing Email message.

Exploring .NET

Each eBook in this series contains a selection of Dan Appleman's technical articles on .NET. Covering virtually every subject in .NET, and written in a variety of styles, you'll find them entertaining as well as educational. Refer to the web site for a complete list of articles.

Introduction to NT/2000 Security Programming with Visual Basic 6

NT Security is a subject that is intimidating, to say the least. But if you dig past the confusing acronyms, you'll find that it's actually very easy to understand. This eBook will help you get started on the right foot with NT security, and give you the foundation of knowledge you'll need to understand even the most obscure security concepts. It will also introduce you to techniques for adding security based features to your applications (with an emphasis on Visual Basic applications).

Also available (print books)**Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed.**

This book includes advanced techniques, in-depth explanations of how and why COM/ActiveX components work the way they do, and how to take full advantage of the capabilities of VB6's component and control development features.

Dan Appleman's Visual Basic Programmer's Guide to the Win32 API

This best-selling reference is what the Windows software development kit would look like if it were written for Visual Basic programmers. With over 80 sample programs, it is your best reference to the core Win32 API. Refer to: www.desaware.com for a complete edition history before you upgrade.

Dan Appleman's Win32 API Puzzle Book & Tutorial for VB Programmers

In this book, Appleman teaches you the skills you'll need to use the 7500+ API functions not covered in his famous *Visual Basic Programmer's Guide to the Win32 API*. You'll learn how to interpret the Microsoft documentation and create declarations for even the most difficult functions.

Appendix C - Publishing

The Story Behind Apress

Apress is an innovative publishing company devoted to meeting the needs of programming professionals and potential programming professionals. Simply put, the *A* in Apress is there because Apress is the *Author's Press*TM. Our author-centric approach to publishing grew from conversations between Dan Appleman and Gary Cornell, whose books are widely considered among the best in the areas they cover. They wanted to create a publishing company that emphasized quality above all—a company whose books might not always be the first to market, but would always be the best to market.

To accomplish this goal, they knew it was necessary to attract the very best authors—established authors whose work is already highly regarded, and new authors who also have the real-world, practical experience professional software developers want in the books they buy. And, this is where the author-centric nature of Apress proves it's worth. We think if you visit our Web site at: www.apress.com, you'll see that Dan and Gary's vision of an author centric press has already attracted many leading software professionals to write for Apress.

Would you like to write for Apress?

Apress is rapidly expanding its publishing program. If you can write and refuse to compromise on the quality of your work; if you believe in doing more than rehashing existing documentation; if you are looking for opportunities and rewards that go far beyond those offered by traditional publishing houses, we want to hear from you!

Consider these innovations that we offer every one of our authors:

- Top royalties with *no* hidden switch statements. For example, authors typically only receive half of their normal royalty rate on foreign sales. In contrast, Apress' royalty rate remains the same for both foreign and domestic sales.
- A share in the wealth. Apress believes that once a book sells enough copies to break even the author as well as the publisher should enjoy the gravy train. Apress contract offers an automatic jump in royalties once a certain number of copies are sold.
- Serious treatment of the technical review process. Each Apress book has a technical reviewing team whose remuneration depends in part on the success of the book since they, too, receive a royalty.

All Apress editors are writers and programmers like yourself. At Apress you'll work with professionals who truly understand your needs and the needs of you readers.

Moreover, through a partnership with Springer-Verlag, one of the world's major publishing houses, Apress has significant distribution power and capital. Thus, we have the resources both to produce the highest quality books *and* to market them aggressively.

If you fit the model of the Apress author then please contact our editorial directors:

editorial@apress.com